USENIX

Proceedings of the 2nd USENIX Conference on File and Storage Technologies

San Francisco, CA, Mar. 31–Apr. 2, 2003

2nd USENIX Conference on File and Storage Technologies

San Francisco, California, USA
March 31–April 2, 2003

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 1-931971-07-2

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)

March 31–April 2, 2003
San Francisco, California, USA

Conference Organizers

Program Chair

Jeff Chase, *Duke University*

Program Committee

Khalil Amiri, *IBM Research*
Remzi Arpaci-Dusseau, *University of Wisconsin, Madison*
Peter Chen, *University of Michigan*
Peter Corbett, *Network Appliance*
Mike Franklin, *University of California, Berkeley*
Eran Gabber, *Lucent Technologies, Bell Labs*
Greg Ganger, *Carnegie Mellon University*
Peter Honeyman, *CITI, University of Michigan*
Frans Kaashoek, *Massachusetts Institute of Technology*
Darrell Long, *University of California, Santa Cruz*
Erik Riedel, *Seagate Research*
Margo Seltzer, *Harvard University*
Keith A. Smith, *Sun Microsystems*
Chandu Thekkath, *Microsoft Research*
John Wilkes, *Hewlett-Packard Labs*

The USENIX Association Staff

External Reviewers

Asker Bazen
Olivier Benoit
Andreas Bogk
Liqun Chen
Jordan C N Chong
Arnaud Contes
Ricardo Corin
Jean-Sébastien Coron
Benoit Gonzalvo
Jochen Haller
Ferdy Hanssen
Keith Harrison
Karl E. Hartel
Ludovic Henrio
Jaap-Henk Hoepman
Fabrice Huet
Günter Karjoth
Jeroen Keuning
Markus Kuhn
Law Yee Wei
Stefan Lucks
Eric Madelaine
Wenbo Mao
Antoni Martínez-Ballesté
Henk Muller
Heiko Oester
Francis Olivier
Béatrice Peirani
David Plaquin
Stéphanie Porte
Florence Quès
Michael Roe
Michael Rohs
Ludovic Rousseau
Torsten Schütze
Francesc Sebé
Nicolas Sendrier
Bernard Paul Serpette
Vasughi Sundramoorthy
Assia Tria
Michael Tunstall
Julien Vayssière
Lionel Victor
Harald Vogt
Peter Windirsch

FAST '03: 2nd USENIX Conference on File and Storage Technologies

March 31–April 2, 2003
San Francisco, CA, USA

Index of Authors

Message from the Program Chair

Welcome to the Second USENIX Conference on File and Storage Technologies (FAST '03). This FAST builds on the success of the first FAST conference last spring, initiated by Darrell Long and other prominent storage system researchers from academia and industry.

The FAST '03 program includes 18 technical papers selected from a pool of 67 submissions. These papers represent some of the outstanding work in the area, ranging from RAID design to secure wide-area file sharing. An important element of the program is a technology session featuring measurement studies to help guide future research and development. The program also features three invited sessions highlighting research trends and industry directions. These sessions support FAST's role as a technology-focused forum that encourages the exchange of ideas between industry and academic researchers.

This year's FAST was blessed with a hard-working and thorough Program Committee of 16 leading researchers, including representatives from many of the companies and industry labs most active in this area. The PC's work was complemented by 50 external reviewers. We joined OSDI and several other recent major conferences in using Dirk Grunwald's web-based reviewing system, which he hosted for us on short notice at the University of Colorado. Dirk's assistance eased our work considerably. His system is an important contribution to the community.

The two-round reviewing process for FAST generated over 300 written reviews. Since many PC members also submitted papers, we handled potential conflicts of interest conservatively. In particular, PC members had no access to the reviewing process for papers authored by their students, postdocs, advisors, home institutions, or recent collaborators. The PC selected the final program in an all-day meeting at Duke University on Saturday, October 26. Authors of accepted papers revised their work based on feedback from the reviewing process, with a PC member acting as a 'shepherd'.

I thank all PC members, reviewers, and authors for your hard work during this process. Your contributions will help to build FAST's reputation for technical and editorial quality comparable to the best conferences and journals. I also thank the FAST sponsors, including HP Labs and the Storage Networking Industry Association (SNIA), and the USENIX staff for their skilled, friendly handling of all that is needed to pull off an event like FAST.

Jeffrey S. Chase, ***Duke University***
Program Chair

Pond: the OceanStore Prototype[*]

Sean Rhea, Patrick Eaton, Dennis Geels,
Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz
University of California, Berkeley
{*srhea,eaton,geels,hweather,ravenben,kubitron*}@*cs.berkeley.edu*

Abstract

OceanStore is an Internet-scale, persistent data store designed for incremental scalability, secure sharing, and long-term durability. Pond is the OceanStore prototype; it contains many of the features of a complete system including location-independent routing, Byzantine update commitment, push-based update of cached copies through an overlay multicast network, and continuous archiving to erasure-coded form. In the wide area, Pond outperforms NFS by up to a factor of 4.6 on read-intensive phases of the Andrew benchmark, but underperforms NFS by as much as a factor of 7.3 on write-intensive phases. Microbenchmarks show that write performance is limited by the speed of erasure coding and threshold signature generation, two important areas of future research. Further microbenchmarks show that Pond manages replica consistency in a bandwidth-efficient manner and quantify the latency cost imposed by this bandwidth savings.

1 Introduction

One of the dominant costs of storage today is management: maintaining the health and performance characteristics of data over the long term. Two recent trends can help ameliorate this problem. First, the rise of the Internet over the last decade has spawned the advent of universal connectivity; the average computer user today is increasingly likely to be connected to the Internet via a high-bandwidth link. Second, disk storage capacity per unit cost has skyrocketed; assuming growth continues according to Moore's law, a *terabyte* of EIDE storage will cost $100 US in under three years. These trends present a unique opportunity for file system designers: for the first time, one can imagine providing truly durable, self-maintaining storage to every computer user.

OceanStore [14, 26] is an Internet-scale, cooperative file system designed to harness these trends to provide high durability and universal availability to its users through a two-tiered storage system. The upper tier in this hierarchy consists of powerful, well-connected hosts which serialize changes and archive results. The lower tier, in contrast, consists of less powerful hosts—including users' workstations—which mainly provide storage resources to the system. Dividing the system into two tiers in this manner allows for powerful, well-connected hosts to provide services that demand many resources, while at the same time harnessing the vast storage resources available on less powerful or less well-connected hosts.

The unit of storage in OceanStore is the *data object*, onto which applications map more familiar user interfaces. For example, Pond includes both an electronic mail application and a UNIX file system. To allow for the greatest number of potential OceanStore applications, we place the following requirements on the object interface. First, information must be universally accessible; the ability to read a particular object should not be limited by the user's physical location. Second, the system should balance the tension between privacy and information sharing; while some applications demand the ability to aggressively read- and write-share data between users, others require their data to be kept in the strictest confidence. Third, an easily understandable and usable consistency model is crucial to information sharing. Fourth, privacy complements integrity; the system should guarantee that the data read is that which was written.

With this interface in mind, we designed OceanStore under the guidance of two assumptions. First, the infrastructure is untrusted except in aggregate. We expect hosts and routers to fail arbitrarily. This failure may be passive, such as a host snooping messages in attempt to violate users' privacy, or it may be active, such as a host injecting messages to disrupt some protocol. In aggregate, however, we expect hosts to be trustworthy; specifically, we often assume that no more than some fraction of a given set of hosts are faulty or malicious.

A second assumption is that the infrastructure is constantly changing. The performance of existing communication paths varies, and resources continually en-

[*]Research supported by NSF career award #ANI-9985250, NFS ITR award #CCR-0085899, and California MICRO award #00-049. Dennis Geels is supported by the Fannie and John Hertz Foundation.

Name	Meaning	Description
BGUID	block GUID	secure hash of a block of data
VGUID	version GUID	BGUID of the root block of a version
AGUID	active GUID	names a complete stream of versions

Table 1: *Summary of Globally Unique Identifiers (GUIDs).*

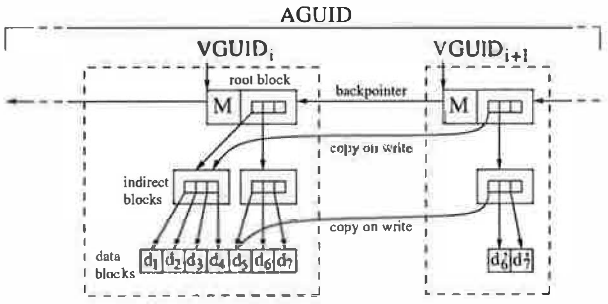


Figure 1: A data object is a sequence of read-only versions, collectively named by an *active* GUID, or AGUID. Each version is a B-tree of read-only blocks; child pointers are secure hashes of the blocks to which they point and are called *block* GUIDs. User data is stored in the leaf blocks. The block GUID of the top block is called the *version* GUID, or VGUID. Here, in version $i + 1$, only data blocks 6 and 7 were changed from version i, so only those two new blocks (and their new parents) are added to the system; all other blocks are simply referenced by the same BGUIDs as in the previous version.

ter and exit the network, often without warning. Such constant flux has historically proven difficult for administrators to handle. At a minimum, the system must be self-organizing and self-repairing; ideally, it will be self-tuning as well. Achieving such a level of adaptability requires both the redundancy to tolerate faults and dynamic algorithms to efficiently utilize this redundancy.

The challenge of OceanStore, then, is to design a system which provides an expressive storage interface to users while guaranteeing high durability atop an untrusted and constantly changing base. In this paper, we present Pond, the OceanStore prototype. This prototype contains most of the features essential to a full system; it is built on a self-organizing location and routing infrastructure, it automatically allocates new replicas of data objects based on usage patterns, it utilizes fault-tolerant algorithms for critical services, and it durably stores data in erasure-coded form. Most importantly, Pond contains a sufficiently complete implementation of the OceanStore design to give a reasonable estimate of the performance of a full system.

The remainder of this paper is organized as follows. We present the OceanStore interface in Section 2, followed by a description of the system architecture in Section 3. We discuss implementation details particular to the current prototype in Section 4, and in Sections 5 and 6 we discuss our experimental framework and performance results. We discuss related work in Section 7, and we conclude in Section 8.

2 Data Model

This section describes the OceanStore *data model*—the view of the system that is presented to client applications. This model is designed to be quite general, allowing for a diverse set of possible applications—including file systems, electronic mail, and databases with full ACID (atomicity, consistency, isolation, and durability) semantics. We first describe the storage layout.

2.1 Storage Organization

An OceanStore *data object* is an analog to a file in a traditional file system. These data objects are ordered sequences of read-only versions, and—in principle—every version of every object is kept forever. Versioning simplifies many issues with OceanStore's caching and replication model. As an additional benefit, it allows for *time travel*, as popularized by Postgres [34] and the Elephant File System [30]; users can view past versions of a file or directory in order to recover accidentally deleted data.

Figure 1 illustrates the storage layout of a data object. Each version of an object contains metadata, the actual user-specified data, and one or more references to previous versions. The entire stream of versions of a given data object is named by an identifier we call its *active globally-unique identifier*, or AGUID for short, which is a cryptographically-secure hash of the concatenation of an application-specified name and the owner's public key. Including this key securely prevents namespace collisions between users and simplifies access control.

To provide secure and efficient support for versioning, each version of a data object is stored in a data structure similar to a B-tree, in which a block references each child by a cryptographically-secure hash of the child block's contents. This hash is called the *block GUID*, or BGUID, and we define the version GUID, or VGUID, to be the BGUID of the top block. When two versions of a data object share the same contents, they reference the same BGUIDs; a small difference between versions requires only a small amount of additional storage. Because they are named by secure hashes, child blocks are read-only. It can be shown that this hierarchical hashing technique produces a VGUID which is a cryptographically-secure hash of the entire contents of a version [20]. Table 1 enumerates the types of GUIDs in the system.

2.2 Application-specific Consistency

In this section, we describe the consistency mechanisms provided to readers and writers of data objects. We define an *update* to be the operation of adding a new version to the head of the version stream of one or more data objects. In OceanStore, updates are applied atomically and are represented as an array of potential actions each guarded by a predicate. This choice was inspired by the Bayou system [8]. Example actions include replacing a set of bytes in the object, appending new data to the end of the object, and truncating the object. Example predicates include checking the latest version number of the object and comparing a region of bytes within the object to an expected value.

Encoding updates in this way allows OceanStore to support a wide variety of application-defined consistency semantics. For example, a database application could implement optimistic concurrency control with ACID semantics by letting the predicate of each update check for changes in the read set, and if none are found, applying the write set in the update's action. In contrast, the operation of adding a message to a mailbox stored as a data object could be implemented as an append operation with a vacuous predicate. One important design decision in OceanStore was not to support explicit locks or leases on data, and to instead rely on our update model to provide consistency; if necessary, the atomicity of our updates allows locks to be built at the application layer.

Along with predicates over updates, OceanStore allows client applications to specify predicates over reads. For example, a client may require that the data of a read be no older than 30 seconds, it may require the most-recently written data, or it may require the data from a specific version in the past.

3 System Architecture

We now discuss the architecture of the OceanStore system that implements the application-level interface of the previous section. The unit of synchronization in OceanStore is the data object. Consequently, although changes to a particular object must be coordinated through shared resources, *changes to different objects are independent.* OceanStore exploits this inter-object parallelism in order to achieve scalability; adding additional physical components allows the system to support more data objects.

3.1 Virtualization through Tapestry

OceanStore is constructed from interacting resources (such as permanent blocks of storage or processes managing the consistency of data). These resources are *virtual* in that they are not permanently tied to a particular piece of hardware and can move at any time. A virtual resource is named by a *globally unique identifier* (GUID) and contains the state required to provide some service. For caches or blocks of storage, this state is the data itself. For more complicated services, this state involves things like history, pending queues, or commit logs.

Virtualization is enabled by a decentralized object location and routing system (DOLR) called Tapestry [12]. Tapestry is a scalable overlay network, built on TCP/IP, that frees the OceanStore implementation from worrying about the location of resources. Each message sent through Tapestry is addressed with a GUID rather than an IP address; Tapestry routes the message to a physical host containing a resource with that GUID. Further, Tapestry is locality aware: if there are several resources with the same GUID, it locates (with high probability) one that is among the closest to the message source.

Both hosts and resources are named by GUIDs. A physical host *joins* Tapestry by supplying a GUID to identify itself, after which other hosts can *route* messages to it. Hosts *publish* the GUIDs of their resources in Tapestry. Other hosts can then route messages to these resources. Unlike other overlay networks, Tapestry does not restrict the placement of resources in the system. Of course, a node may *unpublish* a resource or *leave* the network at any time.

3.2 Replication and Consistency

A data object is a sequence of read-only versions, consisting of read-only blocks, securely named by BGUIDs. Consequently, the replication of these blocks introduces no consistency issues; a block may be replicated as widely as is convenient, and simply knowing the BGUID of a block allows a host to verify its integrity. For this reason, OceanStore hosts publish the BGUIDs of the blocks they store in Tapestry. Remote hosts can then read these blocks by sending messages addressed with the desired BGUIDs through Tapestry.

In contrast, the mapping from the name of a data object (its AGUID) to the latest version of that object (named by a VGUID), may change over time as the file changes. To limit consistency traffic, OceanStore implements primary-copy replication [10]. Each object has a single *primary replica*, which serializes and applies all updates to the object and creates a digital certificate mapping an AGUID to the VGUID of the most recent version. The certificate, called a *heartbeat*, is a tuple containing an AGUID, a VGUID, a timestamp, and a version sequence number. In addition to maintaining the AGUID to latest VGUID mapping, the primary replica also enforces access control restrictions and serializes concurrent updates from multiple users.

To securely verify that it receives the latest heartbeat for a given object, a client may include a nonce in its signed request; in this case the resulting response from the primary replica will also contain the client's name and nonce and be signed with the primary's key. This procedure is rarely necessary, however, since common applications can tolerate somewhat looser consistency semantics. Our NFS client, for example, only requests new heartbeats which are less than 30 seconds old.

We implement the primary replica as a small set of cooperating servers to avoid giving a single machine complete control over a user's data. These servers, collectively called the *inner ring*, use a Byzantine-fault-tolerant protocol to agree on all updates to the data object and digitally sign the result. This protocol allows the ring to operate correctly even if some members fail or behave maliciously. The inner ring implementation is discussed in detail in Section 3.6. The primary replica is a virtual resource, and can be mapped on a variety of different physical servers during the lifetime of an object. Further, the fact that objects are independent of one another provides maximal flexibility to distribute primary replicas among physical inner ring servers to balance load.

In addition to the primary replica, there are two other types of resources used to store information about an object: archival fragments and secondary replicas. These are mapped on different OceanStore servers from those handling the inner ring. We discuss each in turn.

3.3 Archival Storage

While simple replication provides for some fault tolerance, it is quite inefficient with respect to the total storage consumed. For example, by creating two replicas of a data block, we achieve tolerance of one failure for an addition 100% storage cost. In contrast, *erasure codes* achieve much higher fault tolerance for the same additional storage cost.

An erasure code [2] is a mathematical technique by which a block is divided into m identically-sized *fragments*, which are then encoded into n fragments, where $n > m$. The quantity $r = \frac{m}{n} < 1$ is called the *rate* of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from *any* m fragments. For example, encoding a block using a rate $\frac{1}{2}$ code and $m = 16$ produces 32 fragments, any arbitrary 16 of which are sufficient to reconstruct the original block. Intuitively, one can thus see that erasure encoding produces far higher fault tolerance for the storage used than replication. A detailed analysis confirming this intuition can be found in our earlier work [36]. In the prototype, we use a Cauchy Reed-Solomon code [2] with $m = 16$ and $n = 32$.

Erasure codes are utilized in OceanStore as follows. After an update is applied by the primary replica, all newly created blocks are erasure-coded and the resulting fragments are distributed to OceanStore servers for storage. Any machine in the system may store archival fragments, and the primary replica uses Tapestry to distribute the fragments uniformly throughout the system based on a deterministic function of their fragment number and the BGUID of the block they encode.[1] To reconstruct a block at some future time, a host simply uses Tapestry to discover a sufficient number of fragments and then performs the decoding process.

3.4 Caching of Data Objects

Erasure coding, as we have seen, provides very high durability for the storage used. However, reconstructing a block from erasure codes is an expensive process; at least m fragments must be recovered, and assuming that the fragments of a block were stored on distinct machines for failure independence, this recovery requires the use of m distinct network cards and disk arms.

To avoid the costs of erasure codes on frequently-read objects, OceanStore also employs whole-block caching. To read a block, a host first queries Tapestry for the block itself; if it is not available the host then retrieves the fragments for the block using Tapestry and performs the decoding process. In either case, the host next publishes its possession of the block in Tapestry; a subsequent read by a second host will find the cached block through the first. Thus the cost of retrieval from the archive is amortized over all of the readers. Importantly, reconstructed blocks are only soft state; since they can be reconstructed from the archive at any time (for some cost), they can be discarded whenever convenient. This soft-state nature of reconstructed blocks allows for caching decisions to be made in a locally greedy manner (Pond uses LRU).

For reading a particular version of a document, the technique described in the previous paragraph is sufficient to ensure a correct result. However, often an application needs to read the *latest* version of a document. To do so, it utilizes Tapestry to retrieve a heartbeat for the object from its primary replica. This heartbeat is a signed and dated certificate that securely maps the object's AGUID to the VGUID of its latest version.

OceanStore supports efficient, push-based update of the secondary replicas of an object by organizing them into an application-level multicast tree. This tree, rooted at the primary replica for the object, is called the *dissemination tree* for that object. Every time the primary

[1] While using Tapestry in this manner yields some degree of failure independence between fragments encoding the same block, it is preferable to achieve this independence more explicitly. We have a proposal for doing so [37], but it is not yet implemented.

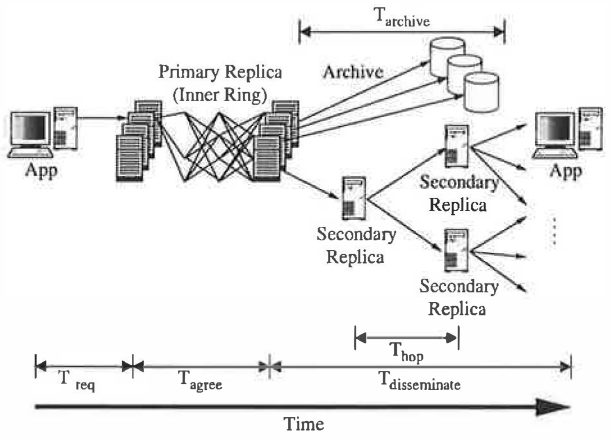


Figure 2: *The path of an OceanStore update.* An update proceeds from the client to the primary replica for its target data object. There, it is serialized with other updates and applied to that target. A heartbeat is generated, certifying the new latest version, and multicast along with the update down the dissemination tree to other replicas. Simultaneously, the new version is erasure-coded and sent to archival storage servers.

replica applies an update to create a new version, it sends the corresponding update and heartbeat down the dissemination tree. Updates are thus multicast directly to secondary replicas. The dissemination tree is built in a self-organizing fashion; each new secondary replica utilizes Tapestry to locate a nearby, pre-existing replica to serve as a parent in this tree. A more sophisticated version of this algorithm is examined elsewhere [5].

3.5 The Full Update Path

In this section, we review the full path of an update. We will postpone the description of the primary replica until the next section.

Figure 2 shows the path of an update in OceanStore. As shown, updates to an object are passed through Tapestry to the primary replica for that object. Once the updates are serialized and committed by the primary replica, they are passed down the dissemination tree to secondary replicas that are currently caching the object. These updates are applied to replicas, thereby keeping them up-to-date. Once updates are applied, they become visible to clients sharing that object. Simultaneous with updating secondary replicas, the primary replica encodes new data in an erasure code, sending the resulting fragments to other OceanStore servers for long-term storage.

Note that Figure 2 illustrates the path of updates for a single object. As shown in Section 6, the process of committing updates by the primary replica is computationally intensive. Thus, it is an important aspect of the system that primary replicas can be distributed among inner ring servers to balance load.

3.6 The Primary Replica

Section 3.2 shows that each data object in OceanStore is assigned an inner ring, a set of servers that implement the object's primary replica. These servers securely apply updates and create new versions. They serialize concurrent writes, enforce access control, check update predicates, and sign a heartbeat for each new version.

To construct this primary replica in a fault-tolerant manner, we adapt a Byzantine agreement protocol developed by Castro and Liskov [4]. Byzantine agreement is a distributed decision process in which all non-faulty participants reach the same decision as long as more than two-thirds of the participants follow the protocol correctly. That is, for a group of size $3f + 1$, no more than f servers may be faulty. The faulty machines may fail arbitrarily: they may halt, send incorrect messages, or deliberately try to disrupt the agreement. Unfortunately, Byzantine agreement requires a number of messages quadratic in the number of participants, so it is infeasible for use in synchronizing a large number of replicas; this infeasibility motivates our desire to keep the primary replicas of an object small in number.

The Castro and Liskov algorithm has been shown to perform quite well in a fault-tolerant network file system. We modify the algorithm for our distributed file system in the following important ways.

Public Key Cryptography: Byzantine agreement protocols require that participants authenticate the messages they send. There are two versions of the Castro-Liskov protocol. In the first version, this authentication was accomplished with public-key cryptography. A more recent version used symmetric-key message authentication codes (MACs) for performance reasons: a MAC can be computed two or three orders of magnitude faster than a public-key signature.

MACs, however, have a downside common to all symmetric key cryptography: they only authenticate messages between two fixed machines. Neither machine can subsequently prove the authenticity of a message to a third party. MACs complicate Castro and Liskov's latter algorithm, but they feel the resulting improvement in performance justifies the extra complexity.

In OceanStore we use aggressive replication to improve data object availability and client-perceived access latency. Without third-party verification, each machine would have to communicate directly with the inner ring to validate the integrity of the data it stores. The computation and communication required to keep each replica consistent would limit the maximum number of copies of any data object–even for read-only data.

We therefore modified the Castro-Liskov protocol to use MACs for all communication internal to the inner ring, while using public-key cryptography to communicate with all other machines. In particular, a digital signature certifies each agreement result. As such, secondary replicas can locally verify the authenticity of data received from other replicas or out of the archive. Consequently, most read traffic can be satisfied completely by the second tier of replicas. Even when clients insist on communicating directly with the ring for maximum consistency, it need only provide a heartbeat certifying the latest version; data blocks can still be sourced from secondary replicas.

Computing signatures is expensive; however, we amortize the added cost of each agreement over the number of replicas that receive the result. Public-key cryptography allows the inner ring to push updates to replicas without authenticating the result for each individually. Also, the increased ability of secondary replicas to handle client requests without contacting the inner ring may significantly reduce the number of agreements performed on the inner ring. We analyze the full performance implications of digital signatures in Section 6.

Proactive Threshold Signatures: Traditional Byzantine agreement protocols guarantee correctness if no more than f servers fail *during the life of the system*; this restriction is impractical for a long-lived system. Castro and Liskov address this shortcoming by rebooting servers from a secure operating system image at regular intervals [4]. They assume that keys are protected via cryptographic hardware and that the set of servers participating in the Byzantine agreement is fixed.

In OceanStore, we would like considerable more flexibility in choosing the membership of the inner ring. To do so, we employ *proactive threshold signatures* [22], which allow us to replace machines in the inner ring without changing public keys.

A threshold signature algorithm pairs a single public key with l private *key shares*. Each of the l servers uses its key share to generate a *signature share*, and any k correctly generated signature shares may be combined by any party to produce a full signature. We set $l = 3f + 1$ and $k = f + 1$, so that a correct signature proves that the inner ring made a decision under the Byzantine agreement algorithm.

A proactive threshold signature scheme is a threshold signature scheme in which a new set of l key shares may be computed that are independent of any previous set; while k of the new shares may be combined to produce a correct signature, signature shares generated from key shares from distinct sets cannot be combined to produce full signatures.

To change the composition of an inner ring, the existing hosts of that ring participate in a distributed algorithm with the new hosts to compute a second set of l shares. These shares are independent of the original set: shares from the two sets cannot be combined to produce a valid signature. Once the new shares are generated and distributed to the new servers, the old servers delete their old shares. By the Byzantine assumption, at most $f = k - 1$ of the old servers are faulty, and the remainder will correctly delete their old key shares, rendering it impossible to generate new signatures with the them. Because the public key has not changed, however, clients can still verify new signatures using the same public key.

A final benefit of threshold signatures is revealed when they are combined with the routing and location services of Tapestry. Rather than directly publishing their own GUIDs, the hosts in the inner ring publish themselves under the AGUIDs of the objects they serve. When the composition of the ring changes, the new servers publish themselves in the same manner. Since the ring's public key does not change, clients of the ring need not worry about its exact composition; the knowledge of its key and the presence of Tapestry are sufficient to contact it.

The Responsible Party: Byzantine agreement allows us to build a fault-tolerant primary replica for each data object. By also using public-key cryptography, threshold signatures, and Tapestry, we achieve the ability to dynamically change the hosts implementing that replica in response to failures or changing usage conditions. One difficulty remains, however: who chooses the hosts in the first place?

To solve this problem, we rely on an entity known as the *responsible party*, so named for its responsibility to choose the hosts that make up inner rings. This entity is a server which publishes sets of failure-independent nodes discovered through offline measurement and analysis [37]. Currently, we access this server through Tapestry, but simply publishing such sets on a secure web site would also suffice. An inner ring is created by selecting one node from each of $3f + 1$ independent sets.

Superficially, the responsible party seems to introduce a single point of failure into the system. While this is true to an extent, it is a limited one. The responsible party itself never sees the private key shares used by the primary replica; these are generated through a distributed algorithm involving only the servers of the inner ring, and new groups of shares are also generated in this manner. Thus, a compromise in the privacy of the data stored by the responsible party will not endanger the integrity of file data. As with primary replicas, there can be many responsible parties in the system; the responsible party thus presents no scalability issue. Furthermore, the online interface to the responsible party only provides the read-only results of an offline computation; there are known solutions for building scalable servers to provide such a service.

4 Prototype

This section describes important aspects of the implementation of the prototype, as well as the ways in which it differs from our system description.

4.1 Software Architecture

We built Pond in Java, atop the Staged Event-Driven Architecture (SEDA) [39], since prior research indicates that event-driven servers behave more gracefully under high load than traditional threading mechanisms [39]. Each Pond subsystem is implemented as a *stage*, a self-contained component with its own state and thread pool. Stages communicate with each other by sending *events*.

Figure 3 shows the main stages in Pond and their interconnections. Not all components are required for all OceanStore machines; stages may be added or removed to reconfigure a server. Stages on the left are necessary for servers in the inner ring, while stages on the right are generally associated with clients' machines.

The current code base of Pond contains approximately 50,000 semicolons and is the work of five core graduate student developers and as many undergraduate interns.

4.2 Language Choice

We implemented Pond in Java for several reasons. The most important was speed of development. Unlike C or C++, Java is strongly typed and garbage collected. These two features greatly reduce debugging time, especially for a large project with a rapid development pace.

The second reason we chose Java was that we wanted to build our system using an event driven architecture, and the SEDA prototype, SandStorm, was readily available. Furthermore, unlike multithreaded code written in C or C++, multithreaded code in Java is quite easy to port. To illustrate this portability, our code base, which was implemented and tested solely on Debian GNU/Linux workstations, was ported to Windows 2000 in under a week of part-time work.

Unfortunately our choice of programming language also introduced some complications; foremost among these is the unpredictability introduced by garbage collection. All current production Java Virtual Machines (JVMs) we surveyed use so-called "stop the world" collectors, in which every thread in the system is halted while the garbage collector runs[2]. Any requests currently being processed when garbage collection starts are stalled for on the order of one hundred milliseconds. Requests that travel across machines may be stopped by several collections in serial. While this event does not happen often, it can add several seconds of delay to a task normally measured in tens of milliseconds.

[2]We currently use JDK 1.3 for Linux from IBM. See http://www.ibm.com/developerworks/java/jdk/linux130/.

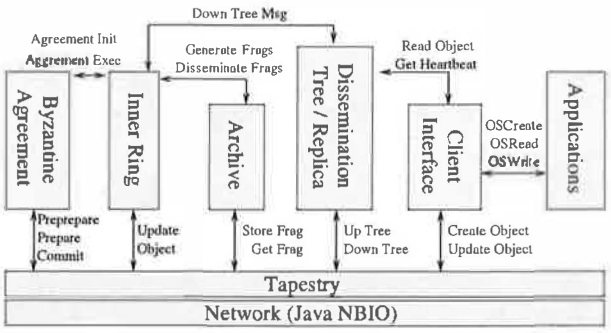


Figure 3: *Prototype Software Architecture.* Pond is built atop SEDA. Components within a single host are implemented as *stages* (shown as boxes) which communicate through events (shown as arrows). Not all stages run on every host; only inner ring hosts run the Byzantine agreement stage, for example.

To adjust for these anomalies, we report the median value and the 0th and 95th percentile values for experiments that are severely effected by garbage collection instead of the more typical mean and standard deviation. We feel this decision is justified because the effects of garbage collection are merely an artifact of our choice of language rather than an inherent property of the system; an implementation of our system in C or C++ would not exhibit this behavior.

4.3 Inner Ring Issues

Most of the core functionality of the inner ring is implemented in Pond, with the following exception. We do not currently implement view changes or checkpoints, two components of the Castro-Liskov algorithm which are used to handle host failure. However, this deficiency should not sufficiently affect our results; Castro and Liskov found only a 2% performance degradation due to recovery operations while running the Andrew500 benchmark [4] on their system.

Lastly, our current signature scheme is a threshold version of RSA developed by Shoup [32]. We plan to implement a proactive algorithm, most likely Rabin's [22], soon; since the mathematics of the two schemes is similar, we expect similar performance from them as well.

5 Experimental Setup

We use two experimental test beds to measure our system. The first test bed consists of a local cluster of forty-two machines at Berkeley. Each machine in the cluster is a IBM xSeries 330 1U rackmount PC with two 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM, and

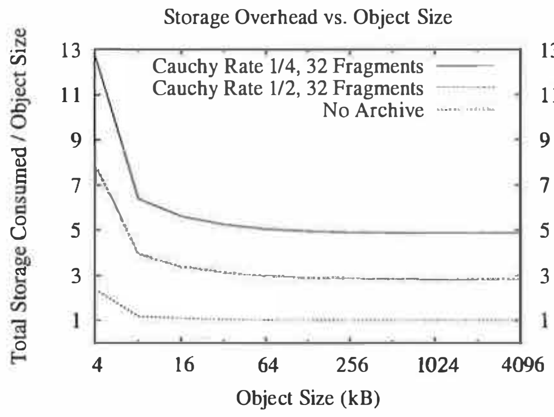


Figure 4: *Storage Overhead.* Objects of size less than the block size of 8 kB still require one block of storage. For sufficiently large objects, the metadata is negligible. The cost added by the archive is a function of the encoding rate. For example, a rate 1/4 code increases the storage cost by a factor of 4.8.

two 36 GB IBM UltraStar 36LZX hard drives. The machines use a single Intel PRO/1000 XF gigabit Ethernet adaptor to connect to a Packet Engines PowerRail gigabit switch. The operating system on each node is Debian GNU/Linux 3.0 (woody), running the Linux 2.4.18 SMP kernel. The two disks run in software RAID 0 (striping) mode using md raidtools-0.90. During our experiments the cluster is otherwise unused.

The second test bed is PlanetLab, an open, global test bed for developing, deploying, and accessing new network services (see http://www.planet-lab.org/). The system currently operates on 101 nodes spread across 43 sites throughout North America, Europe, Australia, and New Zealand. While the hardware configuration of the machines varies slightly, most of the nodes are 1.2 GHz Pentium III CPUs with 1 GB of memory.

For some of our experiments we use a subset of PlanetLab distributed throughout the San Francisco Bay Area in California, USA. The machines that comprise the group of "Bay Area" servers include one machine from each of the following sites: University of California in Berkeley, CA; Lawrence Berkeley National Laboratories in Berkeley, CA; Intel Research Berkeley in Berkeley, CA; and Stanford University in Palo Alto, CA.

6 Results

In this section, we present a detailed performance analysis of Pond. Our results demonstrate the performance characteristics of the system and highlight promising areas for further research.

Key	Update		Update Latency (ms)		
Size	Size	Archive	5%	Median	95%
512	4 kB	off	36	37	38
		on	39	40	41
	2 MB	off	494	513	778
		on	1037	1086	1348
1024	4 kB	off	94	95	96
		on	98	99	100
	2 MB	off	557	572	875
		on	1098	1150	1448

Table 2: *Results of the Latency Microbenchmark in the Local Area.* All nodes are hosted on the cluster. Ping latency between nodes in the cluster is 0.2 ms. We run with the archive enabled and disabled while varying the update size and key length.

6.1 Storage Overhead

We first measure the storage overhead imposed by our data model. As discussed in Section 2, the data object is represented as a B-tree with metadata appended to the top block. When the user data portion of the data object is smaller than the block size, the overhead of the top block dominates the storage overhead. As the user data increases in size, the overhead of the top block and any interior blocks becomes negligible. Figure 4 shows the overhead due to the B-tree for varying data sizes.

The storage overhead is further increased by erasure coding each block. Figure 4 shows that this increase is proportional to the inverse of the rate of encoding. Encoding an 8kB block using a rate $r = \frac{1}{2}$ ($m = 16, n = 32$) and $r = \frac{1}{4}$ ($m = 16, n = 64$) code increases the storage overhead by a factor of 2.7 and 4.8, respectfully. The overhead is somewhat higher than the inverse rate of encoding because some additional space is required to make fragments self-verifying. See [38] for details.

6.2 Update Performance

We use two benchmarks to understand the raw update performance of Pond.

The Latency Microbenchmark: In the first microbenchmark, a single client submits updates of various sizes to a four-node inner ring and measures the time from before the request is signed until the signature over the result is checked. To warm the JVM[3], we update 40 MB of data or perform 1000 updates, depending on the size of the update being tested. We pause for ten seconds to allow the system to quiesce and then perform a number of updates, pausing 100 ms between the response from one update and the request for the next. We report

[3]Because Java code is generally optimized at runtime, the first several executions of a line of code are generally slow, as the runtime system is still optimizing it. Performing several passes through the code to allow this optimization to occur is called *warming* the JVM.

	Time (ms)	
Phase	4 kB Update	2 MB Update
Check Validity	0.3	0.4
Serialize	6.1	26.6
Update	1.5	113.0
Archive	4.5	566.9
Sign Result	77.8	75.8

Table 3: *Latency Breakdown of an Update.* The majority of the time in a small update performed on the cluster is spent computing the threshold signature share over the result. With larger updates, the time to apply and archive the update dominates signature time.

Inner		Avg.	Update	Update Latency (ms)		
Ring	Client	Ping	Size	5%	Median	95%
Cluster	Cluster	0.2	4 kB	98	99	100
			2 MB	1098	1150	1448
Cluster	UCSD	27.0	4 kB	125	126	128
			2 MB	2748	2800	3036
Bay Area	UCSD	23.2	4 kB	144	155	166
			2 MB	8763	9626	10231

Table 4: *Results of the Latency Microbenchmark Run in the Wide Area.* All tests were run with the archive enabled using 1024-bit keys. "Avg. Ping" is the average ping time in milliseconds from the client machine to each of the inner ring servers. UCSD is the University of California at San Diego.

the latency of the median, the fifth percentile, and the ninety-fifth percentile.

We run this benchmark with a variety of parameters, placing the nodes in various locations through the network. Table 2 presents the results of several experiments running the benchmark on the cluster, which show the performance of the system apart from wide-area network effects. This isolation highlights the computational cost of an update. While 512-bit RSA keys do not provide sufficient security, we present the latency of the system using them as an estimate of the effect of increasing processor performance. Signature computation time is quadratic in the number of bits; a 1024-bit key signature takes four times as long to compute as a 512-bit one. The performance of the system using 512-bit keys is thus a conservative estimate of its speed after two iterations of Moore's law (roughly 36 months).

Table 3 presents a breakdown of the latency of an update on the cluster. In the check validity phase, the client's signature over the object is checked. In the serialization phase, the inner ring servers perform the first half of the Byzantine agreement process, ensuring they all process the same updates in the same order. In the update and archive phases, the update is applied to a data object and the resulting version is archived. The final phase completes the process, producing a signed heartbeat over the new version. It is clear from Table 3 that most of the time in a small update is spent computing the

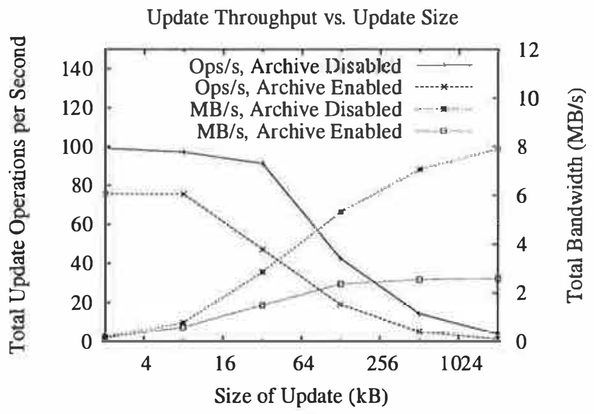


Figure 5: *Throughput in the Local Area.* This graph shows the update throughput in terms of both operations per second (left axis) and bytes per second (right axis) as a function of update size. While the ops/s number falls off quickly with update size, throughput in bytes per second continues to increase. All experiments are run with 1024-bit keys. The data shown is the average of three trials, and the standard deviation for all points is less than 3% of the mean.

threshold signature share over the result. With larger updates, the time to apply and archive the update is large, and the signature time is less important. Although we have not yet quantified the cost of increasing the ring size, the serialize phase requires quadratic communication costs in the size of the ring. The other phases, in contrast, scale at worst linearly in the ring size.

Table 4 presents the cost of the update including network effects. Comparing rows one and two, we see that moving the client to UCSD adds only the network latency between it and the inner ring to the total update time for small updates. Comparing rows two and three we see that distributing the inner ring throughout the Bay Area increases the median latency by only 23% for small updates. Since increased geographic scale yields increased failure independence, this point is very encouraging. For larger updates, bandwidth limitations between the PlanetLab machines prevent optimal times in the wide area; it is thus important that a service provider implementing a distributed inner ring supply sufficient bandwidth between sites.

The Throughput Microbenchmark: In the second microbenchmark, a number of clients submit updates of various sizes to a four-node inner ring. Each client submits updates for a different data object. The clients create their objects, synchronize themselves, and then update the object as many times as possible in a 100 second period. We measure the number of updates completed by all clients and report the update and data throughput.

Figure 5 shows the results of running the throughput

IR Location	Client Location	Throughput (MB/s)
Cluster	Cluster	2.59
Cluster	PlanetLab	1.22
Bay Area	PlanetLab	1.19

Table 5: *Throughput in the Wide Area.* The throughput for a distributed ring is limited by the wide-area bandwidth. All tests are run with the archive on and 1024-bit keys.

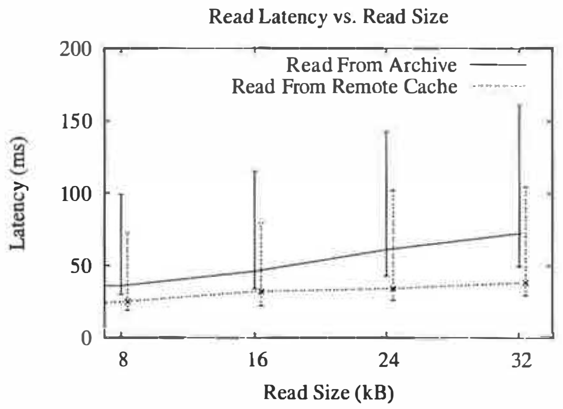


Figure 6: *Latency to Read Objects from the Archive.* The latency to read data from the archive depends on the latency to retrieve enough fragments for reconstruction.

test on the cluster. Again, running the test in the local area illustrates the computational limitations of the inner ring. Lines sloping downward show the number of operations completed per second as a function of the update size and archival policy; lines sloping upward show the corresponding throughput in megabytes per second.

If the inner ring agrees about each update individually, the maximum possible number of operations completed per second is bounded by the speed of threshold signature generation, or approximately 10 operations per second. Instead, the inner ring batches updates and agrees on them in groups (as suggested by [4]); because of this, we have found that the throughput of the system does not change much when using 512-bit keys. Unfortunately, there are other costs associated with each update, so batching only helps to a degree. As suggested by Table 3, however, as the update size increases the signature phase becomes only a small part of the load, so throughput in megabytes per second continues to increase. From Figure 5, we see the maximum throughput of the prototype with the archive disabled is roughly 8 MB/s.

The throughput of the prototype with the archival subsystem enabled is significantly lower. This is not surprising given the effect of the computationally-intensive archiving process we observed in Table 2. From Figure 5, we see that the maximum sustainable throughput of the archival process is roughly 2.6 MB/s. As such, we plan to focus a significant component of our future work on tuning the archival process.

Table 5 shows the results of running the throughput test with the archive running and hosts located throughout the network. In the wide area, throughput is limited by the bandwidth available.

6.3 Archive Retrieval Performance

To read a data object in OceanStore, a client can locate a replica in Tapestry. If no replica exists, one must be reconstructed from archival fragments. The latency of accessing a replica is simply the latency of through Tapestry. Reconstructing data from the archive is a more complicated operation that requires retrieving several fragments through Tapestry and recomputing the data from them.

To measure the latency of reading data from the archive, we perform a simple experiment. First, we populate the archive by submitting updates of various sizes to a four-node inner ring. Next, we delete all copies of the data in its reconstructed form. Finally, a single client submits disjoint read events synchronously, measuring the time from each request until a response is received. We perform 1,000 reads to warm the JVM, pause for thirty seconds, then perform 1,000 more, with 5 ms between the response to each read and the subsequent request. For comparison, we also measure the cost of reading remote replicas through Tapestry. We report the minimum, median, and 95th percentile latency.

Figure 6 presents the latency of reading objects from the archive running on the cluster. The archive is using a rate $r = \frac{m}{n} = \frac{16}{32}$ code; the system must retrieve 16 fragments to reconstruct a block from the archive. The graph shows that the time to read an object increases with the number of 8kB blocks that must be retrieved. The median cost of reading an object from the archive is never more the 1.7 times the cost of reading from a previously reconstructed remote replica.

6.4 Secondary Replication

In this section, we describe two benchmarks designed to evaluate the efficiency and performance of the dissemination tree that connects the second tier of replicas.

The Stream Benchmark: The first benchmark measures the network resources consumed by streaming data through the dissemination tree from a content creator to a number of replicas. We define the efficiency of the tree as the percentage of bytes sent down high-latency links while distributing an update to every replica. We assume that most high-latency links will either have low bandwidth or high contention; local, low-latency links should be used whenever possible.

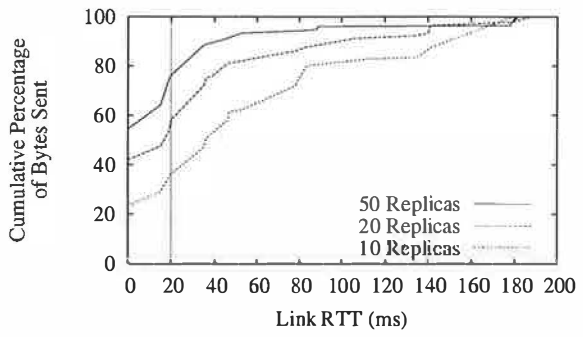


Figure 7: *Results of the Stream Benchmark.* The graph shows the percentage of bytes sent over links of different latency as the number of replicas varies.

In this benchmark, we create a Tapestry network with 500 virtual OceanStore nodes spread across the hosts in 30 PlanetLab sites. We then create a single shared OceanStore data object with a Bay Area inner ring and a variable number of replicas hosted on the seven largest PlanetLab sites. One of these sites lies in the United Kingdom; the other six are in the United States. A single replica repeatedly submits updates that append data to the object. We measure the bandwidth consumed pushing the updates to all other replicas.

Figure 7 shows the percentage of bytes sent across network links of various latencies in this benchmark. According to our metric, the dissemination tree distributes the data efficiently. With only 10 replicas, there are 1.4 replicas per site on average, and 64% of all bytes sent are transmitted across links of latency greater than 20 ms. With 50 replicas, however, there are an average of 7.1 replicas per site, and only 24% of all bytes are sent across links of latency greater than 20 ms.

The Tag Benchmark: The next benchmark measures data sharing in a more interactive scenario, such as a chat room. We arrange for a group of OceanStore replicas to play a distributed game of "tag". To play tag, replicas pass a small piece of data—or *token*—among the group and measure how quickly the token is passed.

In this benchmark, we create a Tapestry network with 200 virtual OceanStore nodes spread across the same 30 PlanetLab sites used in the stream benchmark. We create a single shared data object with a Bay Area inner ring and 50 replicas hosted on the large PlanetLab sites. To pass the token, the replica holding it writes the name of another replica into the data object. A replica receives the token when it reads the new version and finds its name. We measure the average latency over 500 passes.

To put these latencies in perspective, we run two control experiments without using Pond. In these experiments, a coordinator node is placed on one of the machines that hosted an inner ring node in the OceanStore experiment. To pass the token, a replica sends a message to the coordinator; the coordinator forwards the token to the next recipient. In one control experiment, Tapestry is used to communicate between nodes; in the other, TCP/IP is used.

Tokens Passed Using	Latency per Tag (ms)
OceanStore	329
Tapestry	104
TCP/IP	73

Table 6: *Results of the Tag Microbenchmark.* Each experiment was run at least three times, and the standard deviation across experiments was less than 10% of the mean. All experiments are run using 1024-bit keys and with the archive disabled.

As demonstrated by the stream benchmark, the dissemination tree is bandwidth efficient; the tag benchmark shows that this efficiency comes at the cost of latency. Table 6 presents the results of the tag benchmark. In the control cases, the average time to pass the token is 73 ms or 104 ms, depending on whether TCP/IP or Tapestry is used. Using OceanStore, passing the token requires an average of 329 ms. Subtracting the minimum time to perform an update (99 ms, according to Table 4), we see that the latency to pass the token through the dissemination tree is 2.2 times slower than passing the token through Tapestry and 3.2 times slower than using TCP/IP.

6.5 The Andrew Benchmark

To illustrate the performance of Pond on a workload familiar to systems researchers, we implemented a UNIX file system interface to OceanStore using an NFS loopback server [19] and ran the Andrew benchmark. To map the NFS interface to OceanStore, we store files and directories as OceanStore data objects. We use a file's AGUID as its NFS file handle; directories are represented as simple lists of the files that they contain. The information normally stored in a file's inode is stored in the metadata portion of the OceanStore object.

When an application references a file, the replica code creates a local replica and integrates itself into the corresponding object's dissemination tree. From that point on, all changes to the object will be proactively pushed to the client down the dissemination tree, so there is no need to consult the inner ring on read-only operations.

Write operations are always sent directly to the inner ring. NFS semantics require that client writes not be comingled, but imposes no ordering between them. The inner ring applies all updates atomically, so enclosing each write operation in a single update is sufficient to satisfy the specification; writes never abort. Directories must be handled more carefully. On every directory

	LAN			WAN		
	Linux	OceanStore		Linux	OceanStore	
Phase	NFS	512	1024	NFS	512	1024
I	0.0	1.9	4.3	0.9	2.8	6.6
II	0.3	11.0	24.0	9.4	16.8	40.4
III	1.1	1.8	1.9	8.3	1.8	1.9
IV	0.5	1.5	1.6	6.9	1.5	1.5
V	2.6	21.0	42.2	21.5	32.0	70.0
Total	4.5	37.2	73.9	47.0	54.9	120.3

Table 7: *Results of the Andrew Benchmark.* All experiments are run with the archive disabled using 512 or 1024-bit keys, as indicated by the column headers. Times are in seconds, and each data point is an average over at least three trials. The standard deviation for all points was less than 7.5% of the mean.

change, we specify that the change only be applied if the directory has not changed since we last read it. This policy could theoretically lead to livelock, but we expect contention of directory modifications by users to be rare.

The benchmark results are shown in Table 7. In the LAN case, the Linux NFS server and the OceanStore inner ring run on our local cluster. In the WAN case, the Linux NFS server runs on the University of Washington PlanetLab site, while the inner ring runs on the UCB, Stanford, Intel Berkeley, and UW sites. As predicted by the microbenchmarks, OceanStore outperforms NFS in the wide area by a factor of 4.6 during the read-intensive phases (III and IV) of the benchmark. Conversely, the write performance (phases I and II) is worse by as much as a factor of 7.3. This latter difference is due largely to the threshold signature operation rather than wide-area latencies; with 512-bit keys, OceanStore is no more than a factor of 3.1 slower than NFS. When writes are interspersed with reads and computation (phase V), OceanStore performs within a factor of 3.3 of NFS, even with large keys.

7 Related Work

A number of distributed storage systems have preceded OceanStore; notable examples include [31, 13, 8]. More recently, as the unreliability of hosts in a distributed setting has been studied, Byzantine fault-tolerant services have become popular. FarSite [3] aims to build an enterprise-scale distributed file system, using Byzantine fault-tolerance for directories only. The ITTC project [40] and the COCA project [42] both build certificate authorities (CAs) using threshold signatures; the later combines this scheme with a quorum-based Byzantine fault-tolerant algorithm. The Fleet [16] persistent object system also uses a quorum-based algorithm.

Quorum-based Byzantine agreement requires less communication per replica than the state-machine based agreement used in OceanStore; however, it tolerates proportionally less faults. It was this tradeoff that led us to our architecture; we use primary-copy replication [10] to reduce communication costs, but implement the primary replica as a small set servers using state-machine Byzantine agreement to achieve fault tolerance.

In the same way that OceanStore is built atop Tapestry, a number of other peer-to-peer systems are constructing self-organizing storage on distributed routing protocols. The PAST project [28] is producing a global-scale storage system data using replication for durability. The cooperative file system (CFS) [7] also targets wide-area storage. We chose Tapestry for its locality properties; functionally, however, other routing protocols ([17, 18, 23, 27, 33]) could be used instead. Like OceanStore, both PAST and CFS provide probabilistic guarantees of performance and robustness; unlike OceanStore, however, they are not designed for write sharing. Ivy [21] is a fully peer-to-peer read-write file system built atop the CFS storage layer. Unlike OceanStore, it provides no single point of consistency for data objects; conflicting writes must be repaired at the application level. Similarly, Pangaea [29] provides only "eventual" consistency in the presence of conflicting writes. By supporting Bayou-style update semantics and having a single point of consistency per object, OceanStore is able to support higher degrees of consistency (including full ACID semantics) than Ivy or Pangaea; by distributing this single point through a Byzantine agreement protocol, OceanStore avoids losses of availability due to server failures.

Still other distributed storage systems use erasure-coding for durability. One of the earliest is Intermemory [9], a large-scale, distributed system that provides durable archival storage using erasure codes. The Pasis [41] system uses erasure-coding to provide durability and confidentiality in a distributed storage system. Pasis and Intermemory both focus on archival storage, rather than consistent write sharing. Mnemosyne [11] combines erasure-codes with client-directed refresh to achieve durability; clients rewrite data at a rate sufficient to guarantee the desired survival probability.

A final class of systems are also related to OceanStore by the techniques they use, if not in the focus of their design. Publius [15], the Freenet [6], and Eternity Service [1] all focus on preventing censorship of distributed data, each in their own way. Publius uses threshold cryptography to allow a host to store data without knowing its content, as a method of allowing deniability for the host's operators. Freenet also uses coding for deniability, and is built on a routing overlay similar in interface to Tapestry. Finally, the Eternity Service uses erasure coding to make censoring data beyond the resources of any one entity.

8 Conclusions and Future Work

We have described and characterized Pond, the OceanStore prototype. While many important challenges remain, this prototype is a working subset of the vision presented in the original OceanStore paper [14].

Building this prototype has refined our plans for future research. We initially feared that the increased latency of a distributed Byzantine agreement process might be prohibitive, a fear this work has relieved. Instead, threshold signatures have proven far more costly than we anticipated, requiring an order of magnitude more time to compute than regular public key signatures. We plan to spend significant time researching more efficient threshold schemes, or possibly even alternate methods for achieving the benefits they provide. Likewise, we plan to focus on improving the speed of generating erasure-encoded fragments of archival data. Not discussed in this work is the overhead of virtualization. While the latency overhead of Tapestry has been examined before [24], quantifying the additional storage costs it imposes is a topic for future research.

Our future work should not focus entirely on performance, however. One interesting property of the current system is the self-maintaining algorithms it employs. Tapestry automatically builds an overlay network that efficiently finds network resources, and the dissemination tree self-organizes to keep replicas synchronized. The use of threshold signatures allows the inner ring to change its composition without affecting the rest of the system. We hope to make more aspects of the system self-maintaining in the future. For example, algorithms for predictive replica placement and efficient detection and repair of lost data [35] are vital for lowering the management costs of distributed storage systems like OceanStore.

Increased stability and fault-tolerance are also important if Pond is to become a research vehicle for even more interesting projects. Our work in benchmarking Tapestry and its peers [25] was started with the intention of improving the stability of the lowest layer of Pond. Moreover, network partitions are a problem for most overlay networks, and further research is needed to study the behavior of Tapestry under partition. As the stability of Tapestry improves, our focus will shift to higher layers of the system.

Finally, the OceanStore data model has proven expressive enough to support several interesting applications, including a UNIX file system with time travel, a distributed web cache, and an email application. Nonetheless, the developement of these applications has pointed out areas in which the OceanStore API could be improved; a more intuitive API will hopefully spur the developement of further OceanStore applications.

9 Availability

The Pond source code and benchmarks are published under the BSD license and are freely available from http://oceanstore.cs.berkeley.edu.

10 Acknowledgements

We would like thank IBM for providing the hardware in our cluster, and all of the groups that have contributed to PlanetLab. Without these two testbeds, this work would not have been possible. Brent Chun and Mike Howard were particularly helpful with our experiments. Anthony Joseph and Timothy Roscoe provided valuable input on the design and implementation of Pond. Jeremy Stribling's debugging skills were instrumental in bringing up large Tapestry networks. Finally, we are grateful to our anonymous reviewers and Frans Kaashoek, our paper's steward, whose comments and advice have greatly improved this work.

References

[1] R. Anderson. The eternity service. In *Proceedings of Pragocrypt*, 1996.

[2] J. Bloemer et al. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, The International Computer Science Institute, Berkeley, CA, 1995.

[3] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. of Sigmetrics*, June 2000.

[4] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. of OSDI*, 2000.

[5] Y. Chen, R. Katz, and J. Kubiatowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proc. of International Conference on Pervasive Computing*, 2002.

[6] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, July 2000.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[8] A. Demers et al. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, 1994.

[9] A. Goldberg and P. Yianilos. Towards an archival intermemory. In *Proc. of IEEE ADL*, pages 147–156, April 1998.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM SIGMOD Conf.*, June 1996.

[11] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Proc. of IPTPS*, March 2002.

[12] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proc. of ACM SPAA*, pages 41–52, August 2002.

[13] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[14] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[15] A. Rubin M. Waldman and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, 2000.

[16] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *DISCEX II*, 2001.

[17] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of ACM PODC Symp.*, 2002.

[18] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS*, 2002.

[19] D. Mazières. A toolkit for user-level file systems. In *Proc. of USENIX Summer Technical Conf.*, June 2001.

[20] R. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378. Springer-Verlag, 1988.

[21] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[22] T. Rabin. A simplified approach to threshold and proactive RSA. In *Proceedings of Crypto*, 1998.

[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*. ACM, August 2001.

[24] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of INFOCOM*. IEEE, June 2002.

[25] S. Rhea, T. Roscoe, and J. Kubiatowicz. DHTs need application-driven benchmarks. In *Proc. of IPTPS*, 2003.

[26] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, September 2001.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.

[28] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[29] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of OSDI*, 2002.

[30] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of ACM SOSP*, December 1999.

[31] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[32] V. Shoup. Practical threshold signatures. In *Proc. of EUROCRYPT*, 2000.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*. ACM, August 2001.

[34] M. Stonebraker. The design of the Postgres storage system. In *Proc. of Intl. Conf. on VLDB*, September 1987.

[35] H. Weatherspoon and J. Kubiatowicz. Efficient heartbeats and repair of softstate in decentralized object location and routing systems. In *Proc. of SIGOPS European Workshop*, 2002.

[36] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, March 2002.

[37] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of International Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.

[38] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc. of International Workshop on Future Directions of Distributed Systems*, 2002.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM SOSP*, October 2001.

[40] T. Wu, M. Malkin, and D. Boneh. Building intrusion-tolerant applications. In *Proc. of USENIX Security Symp.*, August 1999.

[41] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, August 2000.

[42] L. Zhou, F. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. Technical Report 2000-1828, Department of Computer Science, Cornell University, Ithaca, NY USA, 2000.

Data Staging on Untrusted Surrogates

Jason Flinn[*‡], Shafeeq Sinnamohideen[†‡], Niraj Tolia[†‡], and M. Satyanarayanan[†‡]
[‡]Intel Research Pittsburgh, [*]University of Michigan, and [†]Carnegie Mellon University

Abstract

We show how untrusted computers can be used to facilitate secure mobile data access. We discuss a novel architecture, data staging, *that improves the performance of distributed file systems running on small, storage-limited pervasive computing devices. Data staging opportunistically prefetches files and caches them on nearby surrogate machines. Surrogates are untrusted and unmanaged: we use end-to-end encryption and secure hashes to provide privacy and authenticity of data and have designed our system so that surrogates are as reliable and easy to manage as possible. Our results show that data staging reduces average file operation latency for interactive applications running on the Compaq iPAQ handheld by up to 54%.*

1 Introduction

Can an untrusted and unmanaged computer facilitate secure mobile file access? Surprisingly, the answer is "yes." In this paper, we show how such a computer can be used as a data staging node to improve the performance of cache miss handling in an Internet-wide distributed file system. The untrusted computer, called a *surrogate,* plays the role of a second-level file cache for a mobile client. By prefetching files and staging them on the surrogate, cache misses from a nearby mobile client can be serviced at low latency (typically one wireless hop) instead of full Internet latency.

The use of surrogates for data staging can bridge a growing mismatch between the desires and expectations of users. On the one hand, users want the lightest and smallest wearable or handheld computer—for example, a wristwatch running Linux is no longer a fantasy [23]. On the other hand, users expect productivity improvements from mobile computing; ubiquitous access to personal and project data is a key part of this expectation. A distributed file system can provide such ubiquitous access, but requires crisp handling of cache misses to achieve good interactive performance. For a small file, network latency to a distant file server on the Internet is typically the dominant component of cache miss service time. This can be reduced by redirecting cache misses to data staged on a nearby surrogate while still maintaining the consistency guarantees of the underlying file system. The alternative of totally avoiding cache misses through hoarding [15, 18] is not viable because of limited cache space and the need to view recent updates by other users. Further, it is usually not possible to perfectly predict the set of files that will be accessed when mobile; the working set of files may change unexpectedly in response to real-world events such as phone calls. Consequently, the set of data that may *possibly* be accessed is much larger than the set of data that is *actually* accessed.

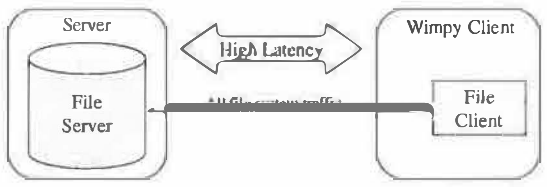
What is the likelihood of a mobile computer finding a nearby surrogate? Although the chances are low today, we predict that continuing decline in mass-market hardware prices will improve these chances in the future. Desktop computers at discount stores already sell for a few hundred dollars, with prices continuing to drop. In the foreseeable future, we envision public spaces such as airport lounges and coffee shops being equipped with surrogates for the benefit of customers, much as comfortable chairs and table lamps are provided today. These will be connected to the wired Internet through high-bandwidth networks, and to mobile clients in their neighborhood through wireless technologies such as 802.11 [13] or Bluetooth [12].

Since hardware cost is only a small part of the total cost of ownership of a system, it is essential that surrogates require virtually no maintenance or system administration. Like a chair or table lamp, they should require negligible attention after initial setup. Only then will they be cheap enough for widespread deployment. This leads to two important assumptions about surrogates in our work: they are *unmanaged* and *untrusted.* In particular, we make surrogates as reliable and easy to manage as possible by maintaining no hard-state on surrogates, building as much as possible on commodity software, and pushing functionality from surrogates to client and server machines.

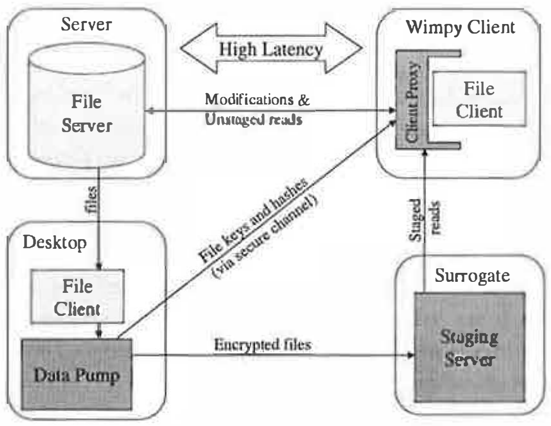
We rely on the concept of *caching trust* to guard against malicious surrogates [26]. This end-to-end approach ensures privacy through encryption, and integrity through verification of secure hashes. Even the most resource-challenged mobile client typically has enough disk or flash storage to cache hashes and encryption keys of all files of potential interest to the user. Hence, data never has to be stored in the clear on a surrogate—it is encrypted before transmission to the surrogate and remains encrypted there. When servicing a cache miss, the client decrypts file data received from the surrogate, calculates the hash, and verifies the computed hash against its cached copy. A compromised surrogate could, of course, still cause mischief through denial of service. The client's only recourse is to contact servers directly. Even in this case, performance is no worse than in the absence of data staging, except for an initial disruption while the client detects that a surrogate is misbehaving and abandons it.

We report on the feasibility of data staging on untrusted surrogates. Our prototype implementation is based on the Coda file system [28], but is structured for easy use with other distributed file systems. Measurements from this prototype confirm the performance benefits of data staging. For bursty, short-term workloads, data staging reduces the cumulative delay due to file operations by up to 54%. We have confirmed these results by replaying long-term traces of file-system activity—these experiments show reductions in file operation latency of up to 49%.

We focus on the file system aspects of data staging. Topics such as surrogate discovery (possibly through mechanisms such as Jini [36] or UPnP [20]) and load balancing across multiple surrogates are part of our plans for future work. We begin, in Section 2, by describing the design and implementation of data staging. The following section describes additional scenarios under which data staging can be profitably employed. We evaluate the benefit of data staging for storage-limited clients in Section 4. The final three sections discuss related work, describe our plans for future work, and summarize our results.



(a) Without data staging



(b) With data staging

Figure 1: Data staging architecture

2 Design and implementation

2.1 Overview

Figure 1(a) shows a typical scenario that motivates the need for data staging. An interactive application running on a storage-limited client accesses files stored in a distributed file system. The file system attempts to reduce access time by caching files on the client machine, but limited space and imperfect prediction prevent it from caching all but a portion of the files that the user might potentially read. Consequently, many files needed by the application are not cached and must be fetched from the distant file server. The user experiences many frustrating delays because the application reads multiple files sequentially and reading each file incurs multiple network round-trips.

Figure 1(b) shows how data staging improves this scenario. On the client machine, we interpose a proxy that intercepts file system traffic. When the proxy observes that remote file accesses are incurring high latency, it finds a surrogate in the nearby network environment that is willing to provide extra storage capacity (currently, this process is manual). The proxy registers with the surrogate and stages the set of files that the user is most likely to access in the future. Since the surrogate has much greater storage capacity than the mobile device, it can store many more files.

Staging is expedited by a *data pump* that typically executes on a user's idle desktop computer located near the file server. When the client proxy wishes to stage a file, it sends a message to the pump through a secure channel. The pump authenticates the message, reads the file from the file system, encrypts the file, and generates a cryptographic hash of the data. The pump transmits the encrypted file to the surrogate and sends the file key and hash to the client through the secure channel. When a staged file is read by the application, the proxy fetches the file from the nearby surrogate, decrypts it, and uses the hash to verify that the file has not been modified. Prefetching files to the surrogate decreases the number of high-latency, blocking file accesses and dramatically reduces the number of long delays experienced by the user.

Reads of unstaged data are serviced using the base file system protocol. The prefetching of files to the surrogate proceeds concurrently with file system traffic. After each file is staged on the surrogate, it becomes immediately available for client use. Thus, as the number of staged files grows, the percentage of cache misses that need to be serviced by the distant file server decreases, leading to significant improvement in interactive application performance. Additionally, the proxy sends all update traffic directly to the file server (although trickle reintegration in Coda may delay updates for a short period of time to improve performance [21]). The client proxy maintains consistency by marking modified files invalid. For workloads with a mixture of read and update traffic, this design improves read performance without compromising security, relaxing consistency guarantees, or significantly delaying updates.

The proxy-based architecture allows us to achieve a great deal of independence from the underlying distributed file system. Data staging requires no modification to file system source code; we use gray-box techniques [3] where necessary to infer file system state. Further, almost all file-system specific code is encapsulated within the client proxy. Thus, while our current system uses Coda, the changes needed to support additional file systems such as NFS and AFS would be minimal.

We first describe the threat model for data staging. In the subsequent three subsections, we describe the design and implementation of the surrogate, client proxy, and data pump in more detail.

2.2 Trust and threat model

Data staging defends against attacks that involve malicious or faulty surrogates. Since we propose that clients opportunistically discover and use third-party surrogates, we place no trust in any surrogate computer. Data staging must defend against attacks that attempt to read private data stored on a surrogate, as well as attacks that corrupt staged data or provide stale data through replay attacks. We also must defend against attacks that attempt to eavesdrop or modify network communication.

Data staging does not explicitly defend against denial-of-service attacks that render surrogates unavailable. Also, a malicious surrogate may periodically refuse to provide requested files to a client. However, if a surrogate performs significantly worse than expected, the client proxy may abandon use of the surrogate without further cost.

Data staging does not defend against attacks that compromise a user's client or desktop machine, or attacks that compromise a file server. We assume that these machines belong to a common administrative domain that enables secure distribution of public keys. Further, we assume that the network communication of the underlying distributed file system is secure. Finally, our protocol assumes that the cryptographic algorithms we employ are sufficiently strong to withstand brute-force attacks.

2.3 Surrogate

2.3.1 Design principles

We are convinced that widespread deployment of surrogates hinges on ease of management. As mentioned previously, surrogates should be as easy to manage as table lamps—they should not need a system administrator or a complex user manual. We have identified three design principles that improve ease of management:

- *Exploit commodity software.* We build as much as possible upon widespread commodity software, so as to leverage the improved reliability that comes through the extensive testing provided by a large user community. To this end, we use the Apache Web server as the base system for our surrogates. We have identified the minimum set of additional functionality that must be located on the surrogate, and provide this functionality with CGI scripts. All other functionality is pushed to the client proxy and data pump in order to keep the custom code base on the surrogate as simple and reliable as possible.
- *Avoid long-term state.* We maintain only soft state on the surrogate so that no critical information is lost if the surrogate is disrupted by power failure or a system crash. For example, we do not buffer client modifications to file data on surrogates. Thus, clients need not guard against malicious or faulty surrogates that might lose modifications. Further, surrogates do not need to run potentially complex reconciliation protocols.
- *Allow file system diversity.* Our surrogate implementation is completely independent of the underlying file system employed by the user. This means that surrogates need not be updated to reflect new file system versions. Further, a single surrogate can simultaneously service users who are employing different underlying file systems.

```
SurrogateRegister     (IN surrogate, IN pubkey, OUT clientid, OUT quota,
                      OUT sesskey, OUT expire);
SurrogateRenew        (IN surrogate, IN clientid, OUT expire);
SurrogateDeregister   (IN surrogate, IN clientid);
SurrogateStage        (IN surrogate, IN clientid, IN fileid, IN buf, IN buflen);
SurrogateUnstage      (IN surrogate, IN clientid, IN fileid);
SurrogateGet          (IN surrogate, IN clientid, IN fileid, IN buf, IN buflen,
                      IN key, IN hash);
```

The surrogate API is implemented as Perl CGI scripts. In total, these scripts consist of 643 lines of source code.

Figure 2: Surrogate API

2.3.2 Surrogate API

Figure 2 shows the surrogate API. Wrapper libraries on the client machine and data pump implement these functions as HTTP/1.1 operations. When `SurrogateGet` is called, the wrapper library issues a HTTP GET request; the remaining functions are implemented as POST operations that invoke CGI scripts on the surrogate.

A client proxy calls `SurrogateRegister` to start using a surrogate. The proxy provides its public key, which is used for authentication. If the surrogate is willing to provide storage space, it assigns the proxy a unique identifier and specifies a storage quota. The surrogate also generates a shared session key, encrypts it with the proxy's public key, and returns it to the proxy. The session key is used to authenticate all subsequent messages that modify surrogate state. `SurrogateRenew`, `SurrogateDeregister`, `SurrogateStage`, and `SurrogateUnstage` each send the surrogate a token encrypted with the session key that represents the command being executed; nonces are used to guard against replay attacks. When a proxy successfully registers, it is granted a lease to use the surrogate. The time duration of the lease is returned by `SurrogateRegister`. Before the lease expires, the proxy may renew it using `SurrogateRenew`.

The `SurrogateStage` function places files on the surrogate. The surrogate treats each file as a binary chunk of data with an identifier unique to the proxy. The CGI script for `SurrogateStage` stores the file data and updates the amount of storage currently used by the proxy. However, if storing the file would cause the proxy to exceed its quota, an error is returned. If a file with the same identifier already exists on the surrogate, the previous data is replaced and the quota updated appropriately.

Once a file is staged, it may be retrieved by the proxy using the `SurrogateGet` function. Alternatively, the proxy may delete the staged file to free up storage capacity with the `SurrogateUnstage` function. The final function, `SurrogateDeregister`, explicitly releases surrogate resources. After a proxy deregisters or its lease expires, the surrogate deletes all files stored on behalf of the proxy.

2.4 Client proxy

The client proxy performs three primary tasks, described further in the next three sections. These tasks are:

- Redirecting file requests to surrogates
- Controlling which files are staged
- Preserving consistency

2.4.1 Redirecting file requests

The client proxy intercepts all traffic bound for a specified set of servers. It masquerades as a local file server; thus, the file system client believes it is connected to a file server running on the local machine. When the proxy receives a request from the file system client, it either transparently forwards it to the distant file server for which it is masquerading, or it retrieves staged data from a nearby surrogate and responds to the request itself. In the presence of multiple file servers, our design allows us to interpose proxies only for high-latency servers—traffic to nearby servers need not incur the additional latency of passing through the proxy.

The proxy maintains a hash table of all files stored on the surrogate. When it intercepts a request to read data from a Coda server, it checks whether a valid copy of the file is currently staged on the surrogate. If the file is staged and valid, it calls `SurrogateGet` to retrieve the data, decrypts the file, computes a secure hash, and verifies that the hash matches its cached value. The file retrieval, decryption, and hash computation are pipelined to reduce access latency. If a valid copy of the file is not staged, the proxy forwards the request to the distant file server.

All modifications to file data on the client machine are sent directly to the file server. If a copy of the modi-

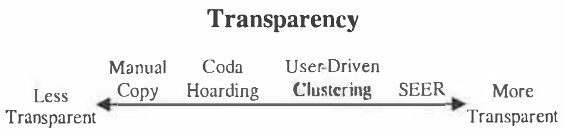


Figure 3: File prediction strategies

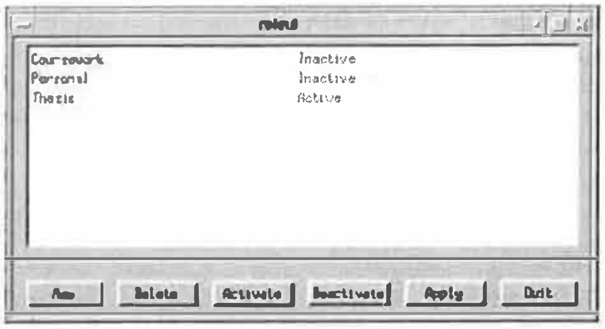
fied file is staged, the proxy marks the copy invalid—this process is described further in Section 2.4.3.

2.4.2 Controlling which files are staged

The client proxy predicts which files are most likely to be accessed in the future and arranges for them to be staged on nearby surrogates. In order to accurately predict future file accesses, it observes all file system traffic. Currently, the proxy takes advantage of Coda's `codacon` interface [27], which reports the name of each file upon invocation of the `open` system call. If the underlying file system did not provide this type of interface, other mechanisms such as the the Sysinternals Filemon tool [32] are available.

Prediction is implemented through a modular interface; whenever a file is opened, the client proxy passes the prediction module the file name, size, and access time. By avoiding a tight integration of the prediction algorithm and the staging implementation, we are able to more easily explore alternative prediction strategies. Figure 3 lists several possibilities, characterizing them by how transparent they are to the user. At one end are completely manual methods, such as explicit copying of files. These methods generate the most distraction; users must specify which files will be needed and must perform the prefetching themselves. The advantage of such methods is better user control of staging. At the other end of the spectrum are completely automated algorithms such as those proposed by Amer [2], Griffioen [11], Kroeger [16], and Kuenning [18]. For example, Kuenning's SEER observes file access patterns and creates clusters of files that are often accessed together. Once a file system determines that some files in a cluster will be accessed, it can prefetch all related files in that cluster.

To date, we have explored two prediction algorithms that lie between manual copy and fully-automated algorithms on the scale of user transparency. Both algorithms use the concept of *roles*; a role is an explicit grouping of files that is associated with a high-level task commonly performed by a user. For example, a graduate student may create roles such as `thesis`, `coursework`, and `personal`. Conceptually, roles are quite similar to SEER's clusters, except that they are externally visible to the user. Roles could potentially be integrated with



Coursework	Inactive
Personal	Inactive
Thesis	Active

Figure 4: User interface for roles specification

higher-level abstractions such as Aura tasks [10, 29] or Lifestreams [9].

The first algorithm is based upon *Coda file system hoarding* [15]. For each role, a user explicitly specifies the set of file system subtrees that are most likely to be accessed. The user also orders these subtrees by expressing a relative priority for prefetching; higher-priority subtrees are more likely to be accessed and will therefore be staged before lower-priority subtrees. Using the interface shown in Figure 4, users specify the set of roles they are currently performing (these are referred to as *active* roles). When the client proxy discovers a surrogate, it creates a prefetch list that is the union of the files specified for all active roles. It stages files in order of priority until its storage quota on the surrogate is exceeded.

The interface in Figure 4 also allows users to cope with unexpected changes in their working set. For example, consider an engineer waiting for a flight in an airport lounge with a nearby staging server. The engineer receives e-mail from a colleague pertaining to a previous joint project. Since this project has unexpectedly become important, the engineer does not have the associated design documents and technical specifications cached. The engineer uses the interface in Figure 4 to activate the role associated with the project. The client proxy then recalculates the priority of files to stage and prefetches related files to the nearby surrogate. As the engineer works on the project, application performance continues to improve as more files are staged.

We call the second algorithm that we explored *user-driven clustering*. This algorithm automatically generates associations between files and roles; the relative priority of files to prefetch is determined through a simple LRU strategy. The prediction algorithm maintains separate LRU lists for each role. Whenever a file is opened, that file is placed at the head of the LRU list for all roles that are currently active. This is a conservative strategy: all files that are part of a role will be in the LRU list, but not all files in the LRU list will be part of a role.

When the client proxy prefetches files to a surrogate, it merges the LRU lists of all active roles, then prefetches the most recently accessed files until its storage quota is exceeded. This strategy avoids the need for users to explicitly specify which files to prefetch, but may potentially be less accurate. Similar to the previous algorithm, users can use the interface in Figure 4 to specify when their working set changes.

Our implementation is based upon two important observations. First, we expect available space on nearby surrogates to change by several orders of magnitude as users move: from zero when no surrogate is available, to gigabytes of storage when a high-capacity surrogate is nearby. Therefore, the amount of prediction information maintained should be independent of the current surrogate quota. We maintain LRU information sufficient to populate the allocated quota of any surrogate we may encounter in the future. While the storage requirements for the LRU data are not large ($<$ 1 MB in our experience), this may still represent a significant portion of the storage capacity of a small handheld. Therefore, we store LRU data in the distributed file system, allowing it to be flushed from the client file system cache when additional storage is needed.

Our second observation is that it is now common for one person to access the same data on multiple machines; for example, a single user may own a home computer, a desktop at work, a laptop, and a handheld device. If the user has recently accessed a file on one machine, it is more likely that the user will soon access the file on other machines. We account for this behavior by storing per-machine LRU data in the distributed file system. When the proxy starts, it combines the LRU data from all machines that the user has recently accessed to generate a global LRU ordering. This ordering is then used to select which files are staged. To minimize update traffic, the client proxy reads and writes LRU information periodically (currently every hour). While this strategy has the potential to lose some data in the event of a system crash, the only effect of such a loss is a decrease in prediction accuracy.

2.4.3 Preserving consistency

The client proxy is the final arbiter of whether data staged on a surrogate is valid. It associates a valid bit with each file in its hash table of staged files. After successfully staging a file, this bit is set to valid. When the proxy intercepts a request that modifies file data, it searches for the file in the hash table and invalidates the entry if found. When another Coda client modifies a staged file, Coda provides a `callback` notification to all Coda clients that have accessed the file. The data pump receives this callback and forwards it to the client proxy. If the modified file is currently staged, the proxy invalidates it.

The proxy periodically rescans the LRU list and recalculates which files should be staged. It stages any file at the head of the LRU list that is marked invalid or is not currently staged. Files further toward the tail of the LRU list are unstaged to make room within the allocated quota. We have chosen to make the set of files staged on the surrogate inclusive with the client's Coda file cache, approximating the stack property. This means that files evicted from the Coda cache are immediately available from the surrogate. The performance penalty of inclusive caching is small, since the surrogate will typically have storage capacity several orders of magnitude greater than that of the client machine.

An alternate approach would be to stage files immediately after they are invalidated or newly created. However, when file modifications are bursty, this alternate approach would lead to many successive stages and invalidations, wasting client bandwidth and energy.

2.5 Data pump

The data pump fetches and stages files on behalf of the client proxy. Although a single data pump could run on the file server, we favor running a data pump on the desktop computer of each user instead. The latter alternative has the benefit of reducing load on file servers; since the desktops have large file caches, most requests to stage data can be serviced without contacting the file server.

Client proxies contact the data pump and establish a secure tunnel for communication. The two parties use public key cryptography to establish a symmetric session key. We use the session key to encrypt all further traffic because symmetric key encryption is less computationally demanding than public key encryption. Public key distribution is simplified since a single user will typically operate both machines (if the pump is located on a desktop), or both machines will lie within the same administrative domain (if the pump is located on a file server).

When the client proxy needs to stage a file, it sends the data pump the file pathname and surrogate IP address. The data pump retrieves the file from the underlying distributed file system, generates a random symmetric key, encrypts the file, and generates a cryptographic hash of the file data. The pump calls `SurrogateStage` to place the encrypted file on the surrogate. If successful, the pump sends the key and hash to the client proxy, which stores them for later reference. Our current implementation uses 64 bit DES encryption and generates 128 bit MD5 digests of file data. The storage requirement of 24 bytes per file is significantly less then the average file size reported in file system studies [7, 35].

The client proxy is multi-threaded, allowing it to overlap computation and network transmission, and to service multiple concurrent requests. As a performance enhancement, the proxy may batch multiple `SurrogateStage` requests into a single HTTP/1.1 POST request; this decreases the time needed to stage a large number of small files.

3 Further benefits of data staging

To date, our work has focused on exploring the benefits of data staging for storage-limited clients in pervasive computing environments. Yet, we believe that data staging will prove desirable in several other important scenarios.

The use of Infostations [37] or Data Blasters [19] has been suggested as a solution for overcoming the bandwidth limitations of wide-area wireless networks. Clients periodically pass through short-range, high-bandwidth zones located within the pervasive infrastructure. For example, such zones may be located near airport gates or at highway tollbooths. Data updates such as file modifications can be burst transmitted to the client during the short period of high-bandwidth connectivity. These data staging scenarios are particularly attractive given advances in ultrawideband (UWB) wireless technology that promise to deliver up to 500 Mb/s within a 5-10 meter radius with minimal energy cost [19]. To utilize this effective bandwidth fully, data must be staged in preparation for burst transmission.

Data staging also has large potential benefits for battery-powered clients. Studies of energy usage show that an 802.11 network interface represents a very large portion of the total energy budget of small handheld devices [8, 31]. Under periods of high network usage, such devices quickly run out of battery power.

Our data staging architecture can significantly extend the battery lifetime of mobile clients in two ways. First, fetching small files from surrogates is significantly faster than fetching the equivalent data from distant file servers. Since the network is active for shorter periods with data staging, the network interface can be put into longer and deeper power saving modes. A second, potentially greater, benefit is that the speculative prefetching of data to clients can be dramatically reduced. With data staging, the latency of a cache miss is much lower. Therefore, the client file system can afford to be less aggressive in keeping the cache up-to-date with the files the user is most likely to fetch. Instead, the client can stage such files on a nearby surrogate, knowing that the penalty of a cache miss will be small.

4 Evaluation

How much does data staging improve the performance of interactive applications running on storage-limited clients?

We answer this question by measuring the performance impact of data staging in two different usage scenarios. In the first scenario, we mirror short-term, bursty activity. We model a user browsing images from a large image library stored in the Coda file system and examine the potential performance benefits of data staging. In the second scenario, we examine the benefits of data staging over a longer time period. We use recorded traces of client file system workloads to represent the activities of a user on a multi-day visit to a distant work location and examine how data staging reduces file operation latency.

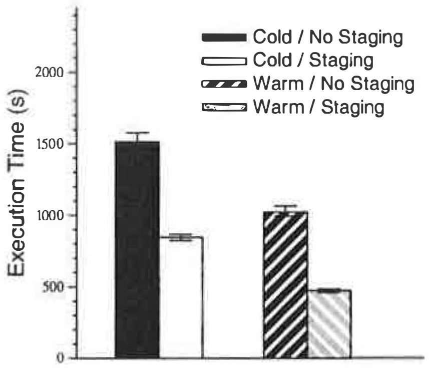
4.1 Experimental setup

We ran a Coda server on a powerful desktop computer with a 2 GHz Pentium 4 processor. We also chose to run the data pump on the same machine since scalability was not a focus of our evaluation. The surrogate ran on an identical desktop computer—we used NISTnet [5] to emulate a 30 ms. delay (60 ms. round-trip time) between the Coda server and surrogate. The 30 ms. delay is typical of current coast-to-coast delays in the United States. The client was a Compaq iPAQ 3850 handheld computer with a 206 MHz StrongArm processor. The iPAQ used a 11 Mb/s 802.11 wireless network card for communication. The wireless hub was on the same network segment as the surrogate; we also emulated a 30 ms. delay between the client and Coda server. All computers ran the Linux 2.4 operating system.

4.2 Image browsing

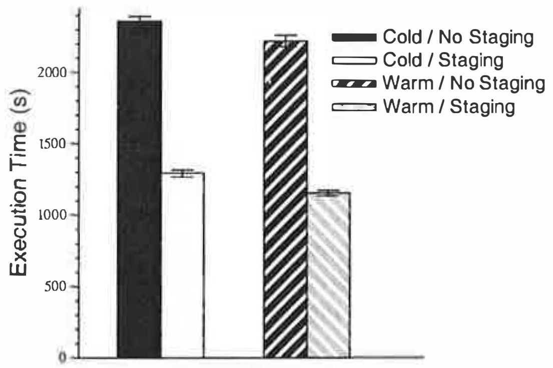
4.2.1 Methodology

Over a roughly one month period, we recorded accesses to a library of digital photographs stored in the Coda file system. The typical client activity captured in this trace first opens a large number of small, thumbnail images in a directory, then opens a smaller number of large, full-sized images in the same directory. From the log, we selected the first 10,148 file operations—these operations read 150 MB of unique file data. However, since many images are read more than once, the total amount of data accessed is 401 MB.



This figure shows the benefits of data staging for the image benchmark with a 64 MB client Coda cache. Each bar is the mean of three trials; the error bars show minimum and maximum values.

Figure 5: Image trace with 64 MB cache



This figure shows the benefits of data staging for the image benchmark with a 16 MB client Coda cache. Each bar is the mean of three trials; the error bars show minimum and maximum values.

Figure 6: Image trace with 16 MB cache

We replayed the trace as fast as possible using the DFS-Trace tool [21], which performs file operations identical to those recorded in the trace. The figure of merit is the time needed to execute the complete trace—this corresponds to the total delay that the user experiences while loading images. This is distinct from the total amount of time the user will take to view the images, which will include a variable amount of think time. The benefits of staging would increase with think time since the client will continue to stage files during such pauses. Without data staging, think time has no noticeable effect upon total delay.

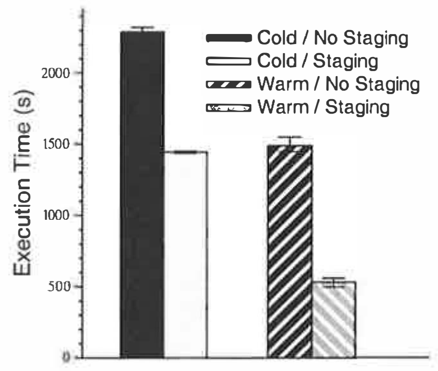
Since the initial state of the client Coda cache will have a large effect on benchmark execution time, we examine the two extreme ends of the spectrum. In the `cold` scenario, no data is contained in the Coda cache when we begin playing a trace; such a scenario will often occur when there is an unexpected change in a user's working set such as the one described in Section 2.4.2. In the `warm` scenario, we fill the entire Coda cache with the set of files initially accessed by the trace; this is the best possible initial cache state.

For both scenarios, we compare total file access time with and without data staging. In the `cold` scenario, the surrogate initially has no data staged. As the trace begins, the client proxy is notified of the change in working set; this emulates the activation of a new role through the interface in Figure 4. Replay of the trace and staging of the data proceed in parallel. Initially, no benefit is derived from data staging; however, as more files are staged, a greater percentage of file accesses are serviced by the surrogate, and average file access latency decreases. Files are staged in random order using a hoard file that lists all files accessed by the trace—the random ordering is a pessimistic assumption, in a production system, LRU or user-specified ordering would cause those files most likely to be accessed to be staged first. For the `warm` scenario, in addition to warming the client Coda cache, we also stage all files referenced by a trace on the surrogate before executing the trace. This represents a scenario where the user has given advance notice of the change in working set sufficient to prefetch all files.

The effectiveness of staging depends upon how closely the set of files staged on the surrogate matches the actual set of files accessed by the client. Inaccuracy is exhibited in two ways. First, the client proxy may stage files that are never accessed—we refer to this as wastage. Specifically, we define the *wastage ratio* to be the ratio of data staged but never accessed to the total amount of data staged. Second, the client proxy may decide not to stage files that it later accesses—we quantify this with a *staging miss ratio*. We calculate the staging miss ratio by dividing the amount of file data accessed by a trace but never staged by the total amount of data accessed by the trace. Our approach to handling this variability is to first choose initial baseline values, specifically a 33% wastage ratio and a 0% staging miss ratio, and then perform sensitivity analysis on each variable.

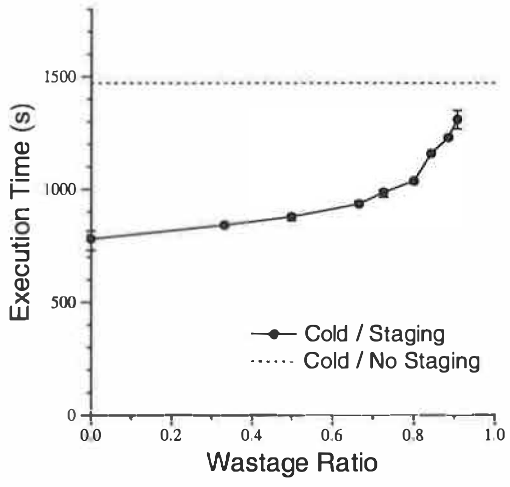
4.2.2 Results

Figure 5 shows the results of running the image benchmark with a 64 MB Coda cache size. In the `cold` scenario, data staging reduces the total time to execute the trace by 44% (11:06 minutes). The `warm` scenario executes in less time because the images initially viewed by the user are already in the Coda cache. In this scenario, the use of data staging reduces execution time by 54% (9:07 minutes).



This figure shows the benefits of data staging for the image benchmark with a 64 MB client Coda cache and a wide-area bandwidth cap of 1 Mb/s. Each bar is the mean of three trials; the error bars show minimum and maximum values.

Figure 7: Image trace with 1 Mb/s bandwidth cap



This figure shows the effect of different wastage ratios on the image trace with a 64 MB client Coda cache. Each data point is the mean of three trials; the error bars show minimum and maximum values.

Figure 8: Effect of wastage on image trace

Figure 6 shows results for a smaller, 16 MB cache size. Somewhat surprisingly, the relative benefits of staging are mostly unaffected by the reduction in cache size. For this trace, much of the short-term file reference locality that is captured by the 64 MB cache size is also captured by the 16 MB cache size. In the `cold` scenario, staging reduces execution time by 45% and, in the `warm` scenario, staging reduces execution time by 48%. However, the absolute magnitude of the benefit increases to 17:46 minutes in the `cold` scenario and 17:44 minutes in the `warm` scenario.

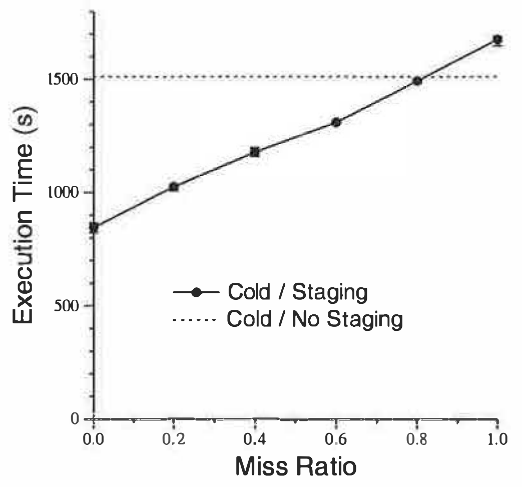
While our work assumes that network bandwidth is not a significant limitation, it is interesting to consider how data staging might perform if wide-area network throughput is limited. Using NISTnet, we capped the maximum throughput of the network link between the pump and surrogate at 1 Mb/s (in addition to imposing a 30 ms. latency). We imposed the same limitation on the link between the client and file server.

Figure 7 shows results for a 64 MB Coda cache size. Since many of the digital photographs are quite large, the bandwidth cap has a significant performance impact. Without data staging, the cap increases trace execution time by 51% in the `cold` scenario and by 46% in the `warm` scenario. In the `cold` scenario, data staging reduces trace execution time by 26%. The bandwidth cap reduces the relative benefit of staging because it takes longer to stage files at the surrogate; a greater percentage of files are accessed directly from the file server. In the `warm` scenario, trace execution time is essentially unaffected by wide-area bandwidth limitations, since all cache misses are serviced by the surrogate. Therefore, data staging is more effective; it reduces trace execution time by 64%. Note that these results reflect wide-area bandwidth limitations. If the bottleneck link is a 1 Mb/s Internet connection shared by client and surrogate, then prefetching traffic may significantly degrade file system performance. Methods that limit prefetching traffic [33] may prove effective in such scenarios.

We next executed the image benchmark with a 64 MB client Coda cache and performed a sensitivity analysis on the wastage ratio. In the `cold` scenario, wastage causes the staging of files accessed in the trace to be delayed. With enough wastage, a file may not be staged until after it is accessed during trace replay. After this point, staging provides no benefit for the file. Wastage does not affect the `warm` scenario since all files are staged before the trace is executed.

As Figure 8 shows, staging reduces the execution time of the image benchmark by 47% in the optimal case where there is no wastage. As wastage increases, the benefits of staging are gradually reduced. With a 91% wastage ratio, staging reduces execution time by only 11%. The effect of wastage may be overstated due to our assumption that files are staged in random order; we expect that if files more likely to be accessed were staged first, wastage would have less effect.

We also examined the effect of different staging miss ratios. We executed the image benchmark with a 64 MB client Coda cache and varied the percentage of files that are contained in the image trace but not staged. We held the wastage ratio constant at 33%. As Figure 9 shows, the benefits of staging decrease as the staging miss ratio grows. With a staging miss ratio of 80%, staging



This figure shows the effect of different staging miss ratios on the image trace with a 64 MB client Coda cache. Each data point is the mean of three trials; the error bars show minimum and maximum values.

Figure 9: Effect of staging miss ratio

Trace	Number of Operations	Length (Hours)	Write Ops.	Working Set (MB)
purcell	87739	55.32	6%	254
messiaen	44027	42.54	2%	227
robin	37504	30.92	7%	85
berlioz	17917	15.70	8%	57

This figure shows the file system traces used for our evaluation. Since data staging currently uses Coda as the base file system and Coda uses the `open-close` semantics of AFS, individual `read` and `write` operations are not included. Hence, "write ops." refers to activities such as `close` after `write` and `mkdir`. The working set is the total size of the files accessed during a trace.

Figure 10: File traces used in evaluation

achieves only a minimal 1% benefit. With a staging miss ratio of 100%, no data is staged. The resulting 11% overhead is the cost of our proxy implementation. Detailed analysis reveals that almost all of this overhead is caused by local remote procedure calls between the client proxy and Coda file system client.

Comparing Figures 8 and 9, it is interesting to note that a higher staging miss ratio reduces the benefit of staging more than the same wastage ratio. Thus, we conclude that prediction strategies should be liberal—if one is uncertain whether or not a file will be needed, it is best to stage it anyway.

4.3 Long-duration file traces

4.3.1 Methodology

To emulate the activity of a user on a multi-day visit to a distant work location, we replayed four traces of client file system activity. We selected these traces, which are summarized in Figure 10, from the set gathered by Mummert et al. [21] at Carnegie Mellon University. Each trace was gathered on a different single-user desktop computer between 1991 and 1993; their durations range from 15 to 55 hours of activity.

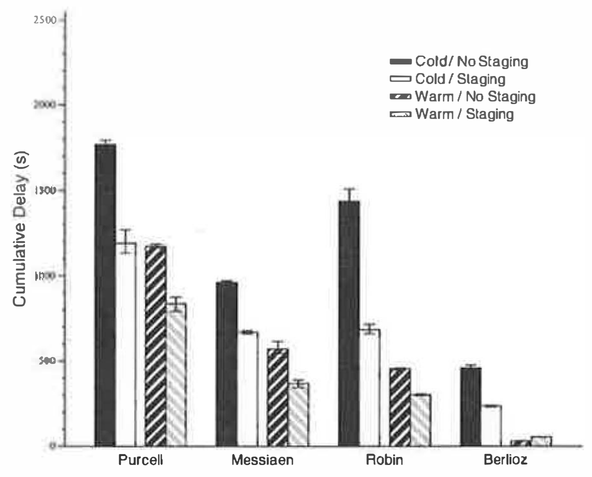
We used DFSTrace to replay each file trace. We repeated the methodology of section 4.2.1 by examining performance with and without data staging in both the `cold` and `warm` scenarios. In both scenarios, we stage the set of files that were captured in the trace using a manually-created hoard file. The order of staging is random in the `cold` scenario. The figure of merit is the total time needed to perform all file operations in the trace; this is equivalent to the amount of delay the user would experience during the trace period while waiting for file operations to complete.

The traces record inter-request delays for file operations. We replay these delays for the first 15 minutes of each trace for the `cold` scenario—during this time, the client proxy stages data on the surrogate while it concurrently proxies file operations. After 15 minutes, staging completes for all traces. From this point on, we eliminate delays and replay the remainder of the trace as fast as possible. This allows us to complete the experiments in a reasonable amount of time. Elimination of delays does not affect the time to service file operations without data staging. With data staging the results are somewhat pessimistic since the client proxy does not restage data that has been invalidated due to modifications.

Although we assume here that all file activity is the result of foreground activity, it is likely that some of the trace activity was generated by background processes; however, this information is very difficult to distinguish from the trace data.

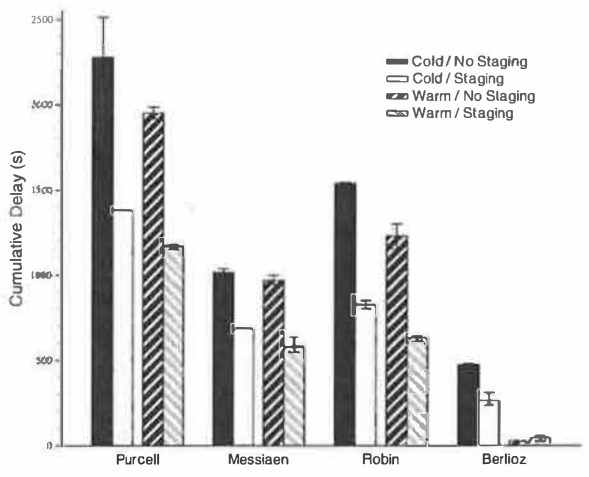
4.3.2 Results

We used an experimental setup identical to the one used for the image benchmark. We first replayed the traces assuming a 64 MB client Coda cache—Figure 11 shows the results. In the `cold` scenario, data staging provides significant benefit for all traces. Staging reduces file operation latency a minimum of 30% for the `messiaen` trace and a maximum of 49% for the `berlioz` trace. In the `warm` scenario data staging causes the `berlioz` trace to take slightly longer to complete. Since the trace's entire working set fits entirely in the Coda cache, there is no need to fetch data from the server—hence, staging provides no benefit. However, staging induces a minimal amount of overhead on each access in order to maintain



This figure shows the benefits of data staging for a 64 MB client Coda cache. Each set of bars shows the cumulative delay due to file system activity for a different trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches. Each bar is the mean of three trials; the error bars show the minimum and maximum values.

Figure 11: File trace results with 64 MB cache



This figure shows the benefits of data staging for a 16 MB client Coda cache. Each set of bars shows the cumulative delay due to file system activity for a different trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches. Each bar is the mean of three trials; the error bars show the minimum and maximum values.

Figure 12: File trace results with 16 MB cache

LRU prediction information; this results in the performance degradation shown in Figure 11. For the other traces, the relative benefit of staging ranges from 29% for the `purcell` trace to 35% for the `messiaen` trace.

Figure 12 shows results when we decrease the Coda cache size to 16 MB. Although the total amount of cumulative delay is larger with a smaller cache size, the relative benefits of staging do not significantly change. In the `cold` scenario, staging reduces cumulative delay from 32% to 46%. In the `warm` scenario, the `berlioz` trace again shows a slight degradation in performance. Even though its working set is larger than the cache size, much of this data is generated during trace replay—16 MB is sufficient to hold almost all data read by the trace. For the remaining traces, staging reduces cumulative delay from 40% to 48%.

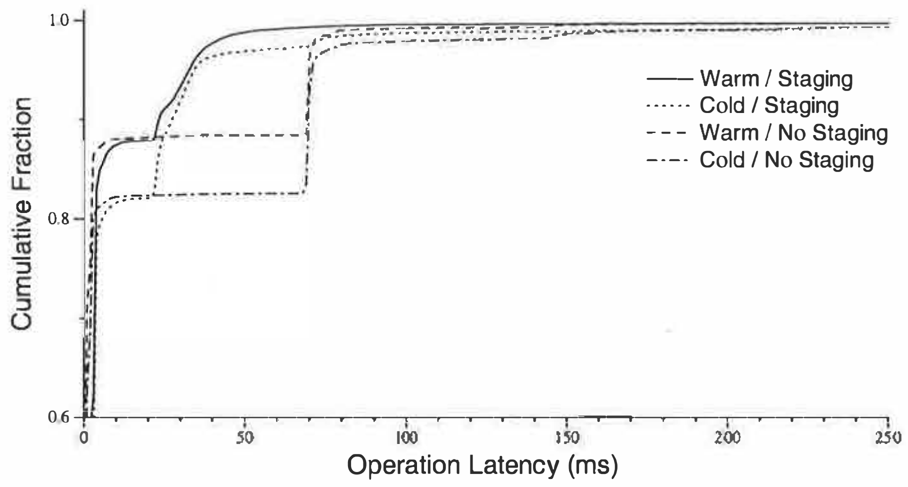
Figure 13 shows more detailed results for a representative trace—`messiaen` with a 64 MB Coda cache. Each line represents the cumulative fraction of file operations that complete within a given time period. For this trace, over 80% of the file operations are reads that hit in the Coda cache or writes that are buffered on the client for later reintegration—these incur negligible delay.

The shape of the `Cold / No Staging` line confirms one of our assumptions: latency, not bandwidth, is the real killer in distributed file system performance. Of the operations that do not complete immediately, the vast majority take slightly more than the 60 ms. round-trip delay. Data staging significantly shortens these high latency operations. For interactive applications that incur the cost of several sequential file operations, this dramatically reduces the frustrating delays experienced by the user.

Finally, we examined the effect of network latency between the client and file server. We expected the benefits of staging to decrease as network round-trip time was reduced. Figure 14 confirms our expectation. When the round-trip delay between client and file server is reduced to 30 ms., data staging decreases cumulative file delay between 5% and 26% in the `cold` scenario. In the `warm scenario`, data staging reduces cumulative file delay by less than 10% for all traces.

5 Future Work

Our current implementation of data staging provides a solid basis for future research. We plan to investigate methods that allow clients to discover servers dynamically. For this purpose, we hope to leverage existing service discovery protocols such as Jini [36] and UPnP [20]. Architectures such as VERSUDS [4] that allow clients to access multiple service discovery protocols through a common interface are especially promising for the heterogeneous environments we support. In addition, the distance-based discovery mechanism proposed by Noble et al. [24] for Fluid Replication may also



This figure shows how data staging reduces file operation latency for the `messiaen` trace with a 64 MB cache. Each line shows the cumulative fraction of file accesses that finish on or before the indicated time.

Figure 13: Reduction of file operation latency for `messiaen` trace

prove to be well-suited for data staging.

We hope to investigate how location prediction can increase the effectiveness of data staging. If the client proxy determines that it will pass near a surrogate in the future, it could proactively stage data there. This functionality is especially useful in constructing Infostation-like environments such as the one described in Section 3.

We plan to support additional file systems; a NFS implementation is in progress. This process is expedited by our encapsulation of file system dependent code in the client proxy and data pump. However, some new issues arise; for instance, without the whole-file caching of Coda, reduction of first-byte latency becomes important. We also plan to investigate the effectiveness of prefetching for different file system parameters (whole file caching vs. block caching, callbacks vs. leases, etc.) Finally, we plan to explore other prediction strategies, especially fully-automated ones such as SEER [18].

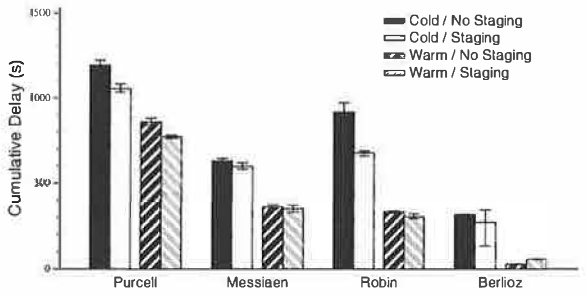
6 Related Work

This work is one of the first to focus on how untrusted and unmanaged hardware can improve distributed file system performance for small, storage-limited clients without compromising security or consistency. In effect, data staging applies the well-understood concept of prefetching [6, 25] to pervasive computing environments. Instead of prefetching file blocks from the disk, data staging prefetches whole files from a distant server.

At a conceptual level, data staging shares several goals with edge computing initiatives such as distributed Web caching. Companies such as Akamai [1] have developed content distribution networks that push data toward end nodes to reduce access latency. However, our focus on file data creates important differences. Consistency is a first-class concern for us: data staging preserves the consistency guarantees of the underlying distributed file system. This is necessary since file system clients are far more likely to modify data than Web clients. Additionally, we provide a mechanism for end-to-end encryption that avoids the need to trust unknown third parties. These issues also differentiate surrogates from caching Web proxies such as Squid [30].

WayStations in Fluid Replication [14] perform a role similar to that played by our surrogates. Yet, there are key differences. First, replicas on WayStations accept file modifications from clients. By transmitting modifications directly to file servers, data staging simplifies the trust model for surrogates. A second difference is that WayStations do not speculatively prefetch data, and thus may underperform in cold-cache scenarios.

OceanStore [17] provides floating replicas of data that can migrate to nearby servers. OceanStore partitions servers into those trusted to perform replication protocols and those not trusted to do so. While the untrusted servers may optimistically accept updates, the client must directly contact the trusted servers in order to ensure permanence and correctness. Thus, replicas on untrusted servers perform similar services to our surrogates. One of the primary differences is our focus on ease of management. We have taken care to place the absolute minimum of functionality on surrogates and have built on commodity software as much as possible—we believe that such steps are vital to ubiquitous surrogate deployment. Another difference is that surrogates are



This figure shows the benefits of data staging for a 64 MB client Coda cache when the network round-trip time is reduced to 30 ms. Each set of bars shows the cumulative delay due to file system activity for a different trace. The first two bars of each data set show the benefit with cold file and surrogate caches; the remaining two bars show the benefit with warm caches. Each bar is the mean of three trials; the error bars show the minimum and maximum values.

Figure 14: File trace results with 30 ms. round-trip time

file-system agnostic. Since all code specific to the underlying file system is isolated in the client proxy and data pump, a single surrogate may simultaneously service clients that employ diverse file systems.

We hope to build on two bodies of related work in the future. First, we hope to incorporate some of the automated prefetching algorithms that have been proposed for file systems [2, 11, 16, 18] and Web distributed cache placement [34]. Second, we plan to use surrogates to implement Infostations [37] that provide high-bandwidth access to data in mobile environments as described in Section 3.

Muntz and Honeyman [22] evaluated the use of intermediate caching for the AFS file system and found that it achieved little benefit. However, their evaluation environment is quite different from the target environment for data staging. Wide-area network latency imposes substantially greater penalties for cache misses.

7 Conclusion

Untrusted and unmanaged machines can facilitate mobile data access. Data staging uses nearby surrogates located in the pervasive computing environment to improve distributed file system performance for storage-limited clients. Clients borrow storage capacity from surrogates and use it as a second-level file cache to hide the latency of file operations.

Two important assumptions in our work are that surrogates are untrusted and unmanaged. Because surrogates are untrusted, we use end-to-end encryption to provide privacy, and secure hashes to ensure authenticity. We make surrogates as reliable and easy to manage as possible by maintaining no hard state on them, using commodity software, and pushing functionality from surrogates to client and server machines. We believe these design considerations will prove vital in ensuring the widespread deployment of a surrogate infrastructure.

8 Acknowledgments

We thank Jan Harkes for his insights into Coda file system behavior and Casey Helfrich for his assistance with setting up our experimental environment. We also thank our shepherd, Peter Honeyman, and the anonymous reviewers for their suggestions.

This research was partly supported by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center (SPAWAR) / U.S. Navy (USN) under contract N66001-99-28918. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements, either express or implied, of DARPA, SPAWAR, USN, Intel, the University of Michigan, Carnegie Mellon University, or the U.S. Government.

References

[1] AKAMAI CORPORATION. *http://www.akamai.com.*

[2] AMER, A., LONG, D. D., AND BURNS, R. C. Group-based management of distributed file caches. In *Proccedings of the 22nd International Conference on Distributed Computing Systems* (Vienna, Austria, July 2002), pp. 525–534.

[3] ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Information and control in gray-box systems. In *Procceddings of the 16th ACM Symp. on Operating Systems Principles* (Banff, Canada, October 2001), pp. 43–56.

[4] BALAN, R., FLINN, J., SATYANARAYANAN, M., SINNAMOHIDEEN, S., AND YANG, H.-I. The Case For Cyber Foraging. In the 10th ACM SIGOPS European Workshop, September 2002.

[5] CARSON, M. *Adaptation and Protocol Testing thorugh Network Emulation.* Internetworking Technologies Group, NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm.

[6] CHANG, F., AND GIBSON, G. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symp. on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 1–14.

[7] DOUCEUR, J., AND BOLOSKY, W. A large-scale study of file-system contents. In *Proceedings of ACM SIGMETRICS* (Atlanta, GA, May 1999), pp. 59–70.

[8] FARKAS, K. I., FLINN, J., BACK, G., GRUNWALD, D., AND ANDERSON, J. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of ACM SIGMETRICS* (Santa Clara, CA, June 2000).

[9] FREEMAN, E., AND GELERNTER, D. Lifestreams: A storage model for personal data. *SIGMOD Record 25*, 1 (1996).

[10] GARLAN, D., SIEWIOREK, D., SMAILAGIC, A., STEENKISTE, P. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing 1*, 2 (April-June 2002).

[11] GRIFFIOEN, J., AND APPLETON, R. Performance measurements of automatic prefetching. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems* (Sept. 1995).

[12] HAARSTEN, J. C. The Bluetooth radio system. *IEEE Personal Communications 7*, 1 (February 2000), 28–36.

[13] IEEE LOCAL AND METROPOLITAN AREA NETWORK STANDARDS COMMITTEE. *Wireless LAN medium access control (MAC) and physical layer (PHY) specifications.* New York, New York, 1997.

[14] KIM, M., COX, L. P., AND NOBLE, B. D. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (January 2002).

[15] KISTLER, J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[16] KROEGER, T., AND LONG, D. The case for efficient file access pattern modeling. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS VII)* (Rio Rico, AZ, March 1999).

[17] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, November 2000), pp. 190–201.

[18] KUENNING, G., AND POPEK, G. Automated hoarding for mobile computers. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997).

[19] LEEPER, DAVID G. Wireless Data Blaster. *Scientific American* (May 2002).

[20] MICROSOFT CORPORATION. *Universal Plug and Play Forum*, June 1999. http://www.upnp.org.

[21] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles* (Copper Mountain, CO, Dec. 1995).

[22] MUNTZ, D., AND HONEYMAN, P. Multi-level caching in distributed file systems or Your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX* (January 1992), pp. 305–313.

[23] NARAYANASWAMI, C., KAMIJOH, N., RAGHUNATH, M., INOUE, T., CIPOLLA, T., SANFORD, J., SCHLIG, E., VENKITESWARAN, S., GUNIGUNTALA, D., KULKARNI, V., YAMAZAKI, K. IBM's Linux Watch: The Challenge of Miniaturization. *IEEE Computer 35*, 1 (January 2002).

[24] NOBLE, B., FLEIS, B., AND KIM, M. A case for fluid replication. In *Network Storage Symposium* (October 2000).

[25] PATTERSON, R., GIBSON, G., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles* (Copper Mountain, CO, Dec. 1995).

[26] SATYANARAYANAN, M. Caching trust rather than content. *Operating System Review 34*, 4 (October 2000).

[27] SATYANARAYANAN, M., EBLING, M. R., RAIFF, J., BRAAM, P. J., AND HARKES, J. *Coda File System User and System Administrators Manual.* Carnegie Mellon University, http://coda.cs.cmu.edu, 1995.

[28] SATYANARAYANAN, M. The Evolution of Coda. *ACM Transactions on Computer Systems 20*, 2 (May 2002).

[29] SOUSA, J., AND GARLAN, D. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS'02)* (Karlsruhe, Germany, April 2002), pp. 7–20.

[30] *Squid Web Proxy Cache.* http://www.squid-cache.org/.

[31] STEMM, M., AND KATZ, R. H. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Science 80*, 8 (August 1997), 1125–1131.

[32] SYSINTERNALS. *http://www.sysinternals.com.*

[33] VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. TCPNice: a mechanism for background transfers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 329–343.

[34] VENKATARAMANI, A., WEIDMANN, P., AND DAHLIN., M. Bandwidth constrained placement in a WAN. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing* (Aug. 2001).

[35] VOGELS, W. File system usage in Windows NT 4.0. In *Proccedings of the 15th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 93–109.

[36] WALDO, J. The Jini architecture for network-centric computing. *Communications of the ACM 42*, 7 (1999).

[37] WU, G., CHU, C.-W., WINE, K., EVANS, J., AND FRENKIEL, R. WINMAC: A novel transmission protocol for Infostations. In *Proceedings of IEEE VTC* (Houston, TX, 1999).

Plutus: Scalable secure file sharing on untrusted storage

Mahesh Kallahalla[*] Erik Riedel[†] Ram Swaminathan[*] Qian Wang[‡] Kevin Fu[§]

Hewlett–Packard Labs
Palo Alto, CA 94304

Abstract

Plutus is a cryptographic storage system that enables secure file sharing without placing much trust on the file servers. In particular, it makes novel use of cryptographic primitives to protect and share files. Plutus features highly scalable key management while allowing individual users to retain direct control over who gets access to their files. We explain the mechanisms in Plutus to reduce the number of cryptographic keys exchanged between users by using filegroups, distinguish file read and write access, handle user revocation efficiently, and allow an untrusted server to authorize file writes. We have built a prototype of Plutus on OpenAFS. Measurements of this prototype show that Plutus achieves strong security with overhead comparable to systems that encrypt all network traffic.

1 Introduction

As storage systems and individual storage devices themselves become networked, they must defend both against the usual attacks on messages traversing an untrusted, potentially public, network as well as attacks on the stored data itself. This is a challenge because the primary purpose of networked storage is to enable easy sharing of data, which is often at odds with data security.

To protect stored data, it is not sufficient to use traditional network security techniques that are used for securing messages between pairs of users or between clients and servers. Thinking of a stored data item as simply a message with a very long network latency is a misleading analogy. Since the same piece of data could be read by multiple users, when one user places data into a shared storage system, the eventual recipient of this "message" (stored data item) is often not known in advance. In addition, because multiple users could update the same piece of data, a third user may from time-to-time update "the message" before it reaches its eventual recipient. Stored data must be protected over longer periods of time than typical message round-trip times.

Most existing secure storage solutions (either encrypt-on-wire or encrypt-on-disk [40]) require the creators of data to trust the storage server to control all users' access to this data as well as return the data intact. Most of these storage systems cater to single users, and very few allow secure sharing of data any better than by sharing a password.

This paper introduces a new secure file system, *Plutus*, which strives to provide strong security even with an untrusted server. The main feature of Plutus is that all data is stored encrypted and all key distribution is handled in a decentralized manner. All cryptographic and key management operations are performed by the clients, and the server incurs very little cryptographic overhead. In this paper we concentrate on the mechanisms that Plutus uses to provide basic filesystem security features — (1) to detect and prevent unauthorized data modifications, (2) to differentiate between read and write access to files, and (3) to change users' access privileges.

Plutus is an encrypt-on-disk system where all the key management and distribution is handled by the client. The advantage of doing this over existing encrypt-on-wire systems is that we can (1) protect against data leakage attacks on the physical device, such as by an untrusted administrator, a stolen laptop, or a compromised server; (2) allow users to set arbitrary policies for key distribution (and therefore file sharing); and (3) enable better server scalability because most of the computationally intensive cryptographic operations are performed at end systems, rather than in centralized servers.

By using client-based key distribution, Plutus can allow user-customizable security policies and authentication mechanisms. Relative to encrypt-on-wire systems, clients individually incur a higher overhead by taking on the key distribution, but the aggregate work within the system remains the same. Our previous analysis with fi-

[*] {maheshk, swaram}@hpl.hp.com.

[†] Work done while at HP Labs. Current address: Segate Technology, Pittsburgh, PA 15222; erik.riedel@seagate.com.

[‡] Work done while at HP Labs. Current address: Mechanical Engineering Department, Pennsylvania State University, University Park, PA 16802; quw6@psu.edu.

[§] Work done while at HP Labs. Current address: Laboratory for Computer Science, MIT, Cambridge MA 02139; fubob@mit.edu.

legroups [40], which aggregates keys for multiple files, shows that the number of keys that any individual needs can be kept manageable.

Instead of encrypting and decrypting files each time they are exchanged over the network, Plutus pre-computes the encryption only when data is updated; this is a more scalable solution as the encryption and decryption cost is distributed among separate users and never involves the server.

We have built a prototype of Plutus in OpenAFS [37]. This enhancement to OpenAFS retains its original access semantics, while eliminating the need for clients to trust servers. Measurements on this prototype show that strong security is achievable with clients paying cryptographic cost comparable to that of encrypt-on-wire systems, and servers not paying any noticeable cryptographic overhead. Since the cryptographic overhead is shifted completely to the clients, the server throughput is close to that of native OpenAFS. Note that these modifications have no impact on the way end applications access files; they change only the way users set sharing permissions on files.

The rest of the paper is organized as follows. Section 2 describes our threat model and assumptions. Section 3 presents the mechanisms and design of Plutus. Section 4 describes a number of subtle attacks that remain possible and outlines potential solutions, and Section 5 describes protocols for creating, reading and writing files, and revoking users. Section 6 describes the implementation and usage of Plutus, and Section 7 evaluates the prototype. We discuss related work in Section 8 and conclude in Section 9.

2 Threat model

This section discusses the assumptions and threat model of Plutus. This paper will use the terminology introduced previously [40] with *owners* (create data), *readers* (read data), *writers* (write and possibly read data), and *servers* (store data).

2.1 Untrusted servers and availability

In Plutus, we trust servers to store data properly, but not to keep data confidential. While a server in Plutus may attempt to change, misrepresent, or destroy data, clients will detect the malicious behavior.

Cryptography alone, however, cannot prevent destruction of data by a malicious server. Replication on multiple servers can ensure preservation of data even when many of the servers are malicious. Systems such as BFS [7], Farsite [1], OceanStore [25], PASIS [17], PAST [12], and S4 [47] address techniques for secure availability through replication. Though, in this paper, we restrict our focus to securing data on a single untrusted file server, the ideas could be generalized for a set of replicated file servers.

2.2 Trusted client machine

Users must trust their local machine. This is, however, difficult to guarantee: providing for a secure program execution environment in an untrusted computing platform is an open problem. Some previous work aims to securely monitor loaded applications [48] or provide partitioned virtual machines to isolate vulnerabilities [10, 48, 50].

2.3 Lazy revocation

Plutus allows owners of files to revoke other people's rights to access those files. Following a revocation, we assume that it is acceptable for the revoked reader to read unmodified or cached files. A revoked reader, however, must not be able to read updated files, nor may a revoked writer be able to modify the files. Settling for lazy revocation trades re-encryption cost for a degree of security. We elaborate on lazy revocation in Section 3.4.

2.4 Key distribution

We assume that users authenticate each other to obtain relevant keys to read and write data on the disk via a secure channel – we do not introduce new authentication mechanisms in this paper. Furthermore, all these exchanges are carried out on-demand; if users want to read/write a file, they contact the file owner (or possibly other readers/writers) to obtain the relevant key. Keys are never broadcast to all users.

2.5 Traffic analysis and rollback

We do not address the issue of traffic analysis in this paper; that is, we do not make any explicit attempt to obfuscate users' access patterns. However, Plutus does support options to encrypt filenames, and encrypts all I/O requests to the server. Recently SUNDR [32] introduced the notion of a rollback attack, wherein an untrusted server tricks a user into accepting version-wise inconsistent or stale data relative to other users. We defer the discussion of rollback protection to a future work [15].

3 Design

In an encrypted file system, we need techniques to (1) differentiate between readers and writers; (2) prevent destruction of data by malicious writers; (3) prevent known plaintext attacks with different keys for different files; (4) revoke readers and writers; and (5) minimize the number of keys exchanged between users. The following core mechanisms together achieve these functions: filegroups, lockboxes, keys, read-write differentiation, lazy revocation, key rotation, and server-verified writes.

3.1 Filegroups and lockboxes

Plutus groups files (not users) into *filegroups* so that keys can be shared among files in a filegroup without compromising security. Filegroups serve as a file aggregation mechanism to prevent the number of cryptographic keys a user manages from growing proportional to the number of files[1]. Aggregating keys into filegroups has the obvious advantage that it reduces the number of keys that users need to manage, distribute, and receive. This is important if users have to perform all key management and distribution themselves. Key aggregation is also necessary to support semi-online users: as in today's systems, Plutus assumes that users are frequently online, but not always. This means that we need an easy mechanism to let an owner share a group of related files, so that the other user may be able to access the related files even when the owner is not online. Additionally, as described in Section 3.2, we associate a RSA key pair with each filegroup. If files were not aggregated and each file had its own key pair, from the measurements in Section 7, each create operation would incur a 2.5 seconds latency to generate the RSA key pair – in comparison, it takes 2.9 seconds to encrypt/decrypt a 20M file with 3DES.

With filegroups, all files with identical sharing attributes are grouped in the same filegroup and are protected with the same key. This exploits the fact that even though a user typically owns and accesses many files, the number of equivalence classes of files with different sharing attributes is small; this enables multiple files to share the same set of keys.

Using filegroups dramatically reduces the number of keys that a user needs to keep track of and the number of keys users must obtain from other users. In the context of the sharing semantics of current UNIX file systems, if two files are owned by the same owner, the same group, and have the same permission bits, then they are authorized for access by the same set of users. All such files could logically be placed in the same filegroup, and encrypted with the same key.

In general there is no relation between the directory hierarchy and the files in a filegroup, though it may be sometimes convenient to define filegroups based on the set of files in one directory (which is, for instance, how AFS defines access rights). Specifically, two encrypted files from two different directories may belong to the same filegroup. Thus, filegroups can be viewed as an invisible overlay on the directory structure.

[1]A previous study [40] mistakenly attributes the filegroup concept to Cepheus [13] instead of itself.

Filegroups uniquely identify all keys that a user needs to perform an operation on a file. This filegroup information can be located together with the rest of the meta-data about the file, for instance, in the UNIX FFS inode (replacing the group and mode bits), or by adding an entry in the disk vnode in AFS [43].

On the downside, using the same key to encrypt multiple files has the disadvantage that the same key encrypts more data, potentially increasing the vulnerability to known plaintext and known ciphertext attacks. However, this is not an issue if these keys are actually the *file-lockbox* keys, and the real file encryption keys are different for different files. The lockbox can then securely hold the different keys; Section 3.3 explains further.

Filegroups also complicate the process of revoking users' access to files because now there are multiple files that the revoked user could have access to. It is tempting to simplify revocation of users by having one key per file. Though this scheme is seemingly more secure (losing a key compromises one file only), managing these keys is a challenge. At best they can be organized into some sort of hierarchy such that the users have to keep fewer keys securely, but this clearly resembles filegroups. Plutus' solution for this problem is discussed in more detail in Section 3.4.

3.2 Keys

Figure 1 illustrates the different objects in Plutus, and how different keys operate on them. Here we describe the structures; later sections discuss these design decisions in more detail. Every file in Plutus is divided into several blocks, and each block is encrypted with a unique symmetric key (such as a DES key), called a *file-block key*. The lockbox, based on ideas in Cepheus [13], holds the file-block keys for all the blocks of the file and is read and written by *file-lockbox* keys. File-lockbox keys are symmetric keys and are given to readers and writers alike. Alternatively, Plutus could use a single file-block key for all blocks of a file and include an initialization vector. File-lockbox keys are the same for all the files in a filegroup. In order to ensure the integrity of the contents of the files, a cryptographic hash of the file contents is signed and verified by a public-private key pair, which we call *file-verify keys* and *file-sign keys*. The file-sign keys are the same for all the files in a filegroup. As an optimization, a Merkle hash tree [34] is used to consolidate all the hashes, with only the root being signed.

Unlike files, which are encrypted at the block level, entries of directories are encrypted individually. This allows the server to perform space management without involving the clients, such as allocating inodes and performing a fsck after a crash. Also, this allows users to browse directories and then request the corresponding keys from

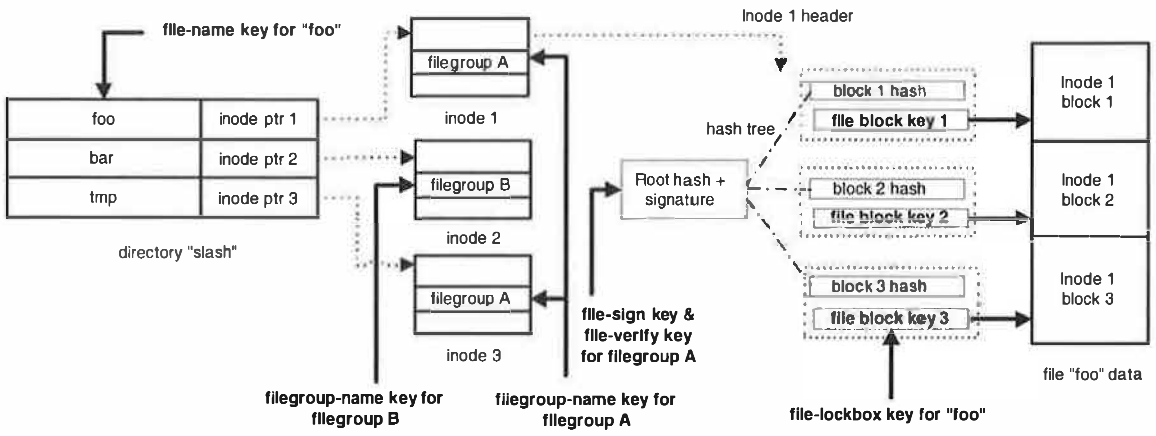


Figure 1: Keys in Plutus. The keys are all highlighted in bold and are linked to the objects that they operate on using bold lines. Dashed lines indicate object pointers. *File-name keys* can encrypt the names of files in directories. An inode contains the names of the filegroup that the file belongs to, and the *filegroup-name key* can encrypt filegroup names. The header contains the Merkle hash tree. The leaves of the hash tree are lockboxes containing the *file-block keys*, which are encrypted with the *file-lockbox key*. The signature of the root is computed and verified using the *file-sign key* and *file-verify key*, respectively.

the file's owner. The filegroup and owner information is located in the inode, as in the case of UNIX. The names of files and filegroups can be encrypted with the *file-name key* and *filegroup-name key*, respectively. Encrypting the names of files and filegroups protects against attacks where the malicious user can glean information about the nature of the file.

All the above described keys are generated and distributed by the owners of the files and filegroups. In addition, currently in Plutus, readers and writers can (re)distribute the keys they have to other users. Plutus intentionally avoids specifying the protocols needed to authenticate users or distribute keys: these are independent of the mechanisms used to secure the stored data and can be chosen by individual users to match their needs.

3.3 Read-write differentiation

One of the basic security functions that file systems support is the ability to have separate readers and writers to the same file. In Plutus, this cannot be enforced by the server as it itself is untrusted; instead we do this by the choice of keys distributed to readers and writers. File-lockbox keys themselves cannot differentiate readers and writers, but can do so together with the file-sign and file-verify key pairs. The file-sign keys are handed to writers only, while readers get the file-verify keys. When updating a data block, a writer recomputes the Merkle hash tree over the (current) individual block hashes, signs the root hash, and places the signed hash in the header of the file. Readers verify the signature to check the integrity of the blocks read from the server. Though using public/private keys for differentiated read/write access was mentioned in the work on securing replicated data [49], the design stopped short of finding a cryptosystem to implement it.

Note that though the file-verify key is same as the public key in a standard public-key system, it is not publicly disseminated. Owners of files issue the file-verify key only to those they consider as authorized readers; similar is the case with the file-sign key.

In our implementation, we use RSA for the sign/verify operations. Then only the readers and writers know N (the RSA modulus). The file-verify key, e, is not a low-exponent prime number (it has to be greater than $N^{1/4}$ [6]). Writers get (d, N), while readers get (e, N).

3.4 Lazy revocation

In a large distributed system, we expect revocation of users to happen on a regular basis. For instance, according to seven months of AFS protection server logs we obtained from MIT, there were 29,203 individual revocations of users from 2,916 different access control lists (counting the number of times a single user was deleted from an ACL). In general, common revocation schemes, such as in UNIX and Windows NT, rely on the server checking for users' group membership before granting access. This requires all the servers to store or cache information regarding users, which places a high trust requirement on the servers and requires all the servers to maintain this authentication information in a secure and consistent manner.

Revocation is a seemingly expensive operation for encrypt-on-disk systems as it requires re-encryption (in

Plutus, re-computing block hashes and re-signing root hashes as well) of the affected files. Revocation also introduces an additional overhead as owners now need to distribute new keys to users. Though the security semantics of revocation need to be guaranteed, they should be implemented with minimal overhead to the regular users sharing those files.

To make revocation less expensive, one can delay re-encryption until a file is updated. This notion of lazy revocation was first proposed in Cepheus [13]. The idea is that there is no significant loss in security if revoked readers can still read unchanged files. This is equivalent to the access the user had during the time that they were authorized (when they could have copied the data onto floppy disks, for example). Expensive re-encryption occurs only when new data is created. The meta-data still needs to be immediately changed to prevent further writes by revoked writers. We discuss subtle attacks that are still possible in Section 4.

A revoked reader who has access to the server will still have read access to the files not changed since the user's revocation, but will never be able to read data updated since their revocation.

Lazy revocation, however, is complicated when multiple files are encrypted with the same key, as is the case when using filegroups. In this case, whenever a file gets updated, it gets encrypted with a new key. This causes filegroups to get fragmented (meaning a filegroup could have more than one key), which is undesirable. The next section describes how we mitigate this problem; briefly, we show how readers and writers can generate all the previous keys of a fragmented filegroup from the current key.

3.5 Key rotation

The natural way of doing lazy revocation is to generate a new filegroup for all the files that are modified following a revocation and then move files to this new filegroup as files get re-encrypted. This raises two issues: a) there is an increase in the number of keys in the system following each revocation; and b) because the sets of files that are re-encrypted following successive revocations are not really contained within each other, it becomes increasingly hard to determine which filegroup a file should be moved to when it is re-encrypted. We address the first issue by relating the keys of the involved filegroups. To address the second issue, we set up the keys so that files are always (re)encrypted with the keys of the latest filegroup; then since keys are related users need to just remember the latest keys and derive previous ones when necessary. We call the latter process *key rotation.*

There are two aspects of rotating the keys of a filegroup a) rotating file-lockbox keys, and b) rotating file-sign and file-verify keys. In either case, to make the revocation secure, the sequence of keys must have the following properties:

a) Only the owner should be able to generate the next version of the key from the current version. This is to prevent anyone from undoing the revocation.

b) An authorized reader should be able to generate all previous versions of the key from the current version. Then readers maintain access to the files not yet re-encrypted, and readers may discard previous versions of the key.

In Plutus, each reader has only the latest set of keys. Writers are directly given the newest version of the keys, since all file encryptions always occur with the newest set of keys. The owners could also do the new-key distribution non-interactively [14], without making point-to-point connections to users.

To assist users in deciding which keys to use, each key has a version number and an owner associated with it. Each file has the owner information, and the version number of the encryption key embedded in the inode. Note that this serves only as a hint to readers and is not required for correctness. Readers can still detect stale keys when the block fails to pass the integrity test.

Next we will describe how we achieve key rotation for file-lockbox keys and file-sign/file-verify keys.

3.5.1 Rotating file-lockbox keys

Whenever a user's access is revoked, the file owner generates a new version of the file-lockbox key. For this discussion, let v denote the version of the file-lockbox key. The owner generates the next version file-lockbox key from the current key by exponentiating the current key with the owner's private key (d, N): $K_{v+1} = K_v^d \bmod N$. This way only the owner can generate valid new file-lockbox keys.

Authorized readers get the appropriate version of the file-lockbox key as follows. (Figure 2 illustrates the relation between the different file-lockbox key versions.) Let w be the current version of the file-lockbox key that a user has.

- If $w = v$ then the reader has the right file-lockbox key to access the file.
- If $w < v$ then the reader has an older version of the key and needs to request the latest file-lockbox key from the owner.
- If $w > v$ then the reader needs to generate the older version of the file-lockbox key using the following recursion. If K_w is the file-lockbox key associated with version w, then $K_{w-1} = K_w^e \bmod N$, where (e, N) is the owner's public key. Readers can recursively generate all previous file-lockbox key from the current key.

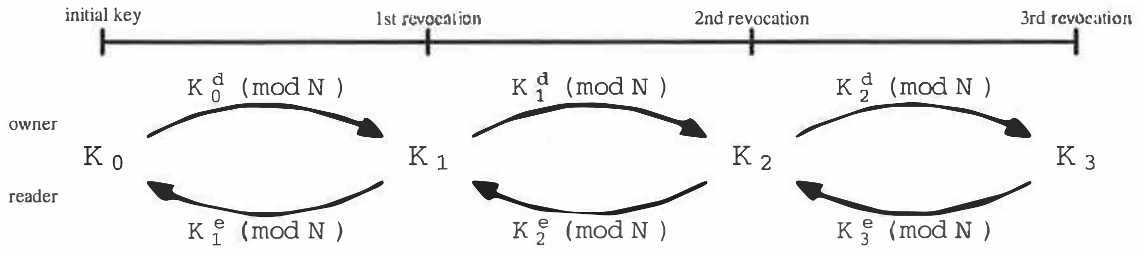


Figure 2: Key rotation for file-lockbox keys. Using RSA, an owner can rotate a key K_i forward. Users can only rotate keys backwards in time. (e, N) is the owner's public key and (d, N) is the owner's private key

In the above protocol, we use RSA encryption as a pseudorandom number generator; repeated encryption is not likely to result in cycling, for otherwise, it can be used to factor the RSA modulus N [33]. Though we use RSA for our key rotation, the property we need is that there be separate encryption and decryption keys, and that the sequence of encryptions is a pseudorandom sequence with a large cycle; most asymmetric cryptosystems have this property.

Though this scheme resembles Lamport's password scheme [27], our scheme is more general. Our scheme provides for specific users (owners) to rotate the key forward, while allowing some other users (readers) to rotate keys backwards.

3.5.2 Rotating file-sign and file-verify keys

By using the file-lock box key generated above as a seed, we can bootstrap the seed into file-sign and file-verify keys as follows. Let the version v file-sign key be (e_v, N_v) and the corresponding file-verify key be (d_v, N_v). In Plutus N_v is stored in file's header in the clear, signed by the owner to protect its integrity. Note that all files in the file-group with the same version have the same value for N_v.

When a user is revoked, the owner picks a new RSA modulus N_v, and rotates the file-lockbox key forward to K_v. Using the latest seed K_v, owners and readers generate the file-verify key as follows. Given the seed K_v, e_v is calculated by using K_v as a seed in a pseudo-random number generator. The numbers output are added to $\sqrt{N_v}$ and tested for primality. The first such number is chosen as e_v. The conditions that $e_v \geq \sqrt{N_v}$ and e_v is a prime guarantee that $\gcd(e_v, \phi(N_v)) = 1$ [28], making it a valid RSA public key. (Notice that the latter test cannot be performed by readers because they do not know $\phi(N_v)$). The pair (e_v, N_v) is the file-verify key.

Owners generate the corresponding RSA private key d_v and use it as the file-sign key. Since writers never have to sign any data with old file-sign keys, they directly get the latest file-sign key (d_v, N_v) from the owner. If the writers have no read access, then they never get the seed, and so it is hard for them to determine the file-verify key from the file-sign key.

Given the current version seed K_v, readers can generate previous version file-verify keys (d_u, N_u), for $u < v$ as follows. They first rotate the seed K_v backwards to get the seed K_u, as described in the previous section. They read (and verify) the modulus N_u from the file header. They then use the procedure described above to determine e_u from N_u and K_u.

The reason for changing the modulus after every revocation is to thwart a subtle collusion attack involving a reader and revoked writer – if the modulus is fixed to, say N', a revoked writer can collude with a reader to become a valid writer (knowing e_v, d_v, and N' allows them to factor N', and hence compute the new file-sign key).

3.6 Server-verified writes

The final question we address is how to prevent unauthorized writers from making authentic changes to the persistent store. Because the only parties involved in the actual write are the server and the user who wishes to write, we need a mechanism for the server to validate writes.

In traditional storage systems, this has been accomplished using some form of an access control list (ACL); the server permits writes only by those on the ACL. This requires that the ACL be stored securely, and the server authenticates writers using the ACL.

In Plutus, a file owner stores a hash of a write token stored on the server to validate a writer. This is semantically the same as a shared password.

Suppose a filename F is not encrypted. The owner of the file creates a write-verification key K_w as the write token. Then, F and the hash of the write token, $H[K_w]$, are stored on the server in the file's inode.

Upon authentication, writers get the write token K_w from the owner. When writers issue a write, they pass the token to the server. To validate the write, the server can now compute $H[K_w]$ and compare it with the stored value. Readers cannot generate the appropriate token because they do not know K_w. The token is secure since the hash value is stored only on the server. Optionally the server can cache the hashed write tokens to speed up write verification.

One problem with the above scheme is that from $H[K_w]$, anyone can learn useful structural information such as which files belong to which filegroup even when the filegroup name is encrypted. This is undesirable given that storage system itself can be stolen and it does not do any authentication of the readers. Such attacks can be thwarted by replacing $H[K_w]$ with $H[K_w, F]$ and the filegroup name with $H[K_g, F]$, where K_g is the filegroup-name key.

The write token used above is similar to the capabilities used in NASD [19] and many systems before [29]. However capabilities in general are given out by a centralized server whereas write tokens are generated by individual file owners and are given to writers in a distributed manner.

The benefit of this approach is that it allows an untrusted server to verify that a user has the required authorization, without revealing the identity of the writer to the server. The scheme also makes it easy for the server to manage the storage space by decoupling the information required to determine allocated space from the data itself. Though the actual data and (possibly) filenames and filegroup names are encrypted and hidden, the list of physical blocks allocated is visible to the server for allocation decisions.

There are several file systems such as Cedar [18], Elephant [41], Farsite [1], Venti [38], and Ivy [36], which treat file data as immutable objects. In a cryptographic storage file system with versioning, server-verified writes are less important for security. Readers can simply choose to ignore unauthorized writes, and servers need worry only about malicious users consuming disk space. In non-versioning systems, a malicious user could corrupt a good file, effectively deleting it.

4 Security analysis

This section explores the set of attacks that remain possible and explains how to adapt Plutus to thwart these attacks. We also argue that some of the remaining attacks can never be handled within the context of our system at any reasonable additional cost.

In decreasing order of severity, an attacker may:

(a) write new data with a new key
(b) write new data with an old key
(c) write old data with an old key; that is, revert to an old version
(d) destroy data
(e) read updated data
(f) read data that has not yet been updated.

These attacks can be prevented by some combination of the following mechanisms: change the read/write verification token (T), re-encrypt the lockbox with a new key (L), and re-encrypt the file itself with a new key (D). Table 1 presents the possible attacks classified into those that a revoked reader could mount, or those that a revoked writer could mount. In each case, the attacker may act alone or in collusion with the server. The attacks that writers can mount depend upon whether an unsuspecting reader has the updated keys or not.

If a system uses lazy revocation, we can prevent revoked readers from accessing data that has been updated. However to prevent them from accessing data that has not been updated, we would need some form of "read verification" — verification of read privileges on each read access, analogous to write-verification. If this verification were done by the storage server then the reader could not get to the data alone, but could do so in collusion with the server. To prevent this attack, the file must be re-encrypted, re-encrypting just the lockbox would be insufficient.

The problem with revoked writers is more severe. Again, we can prevent revoked writers from updating data by verifying each write. But if this verification is done by the server – as in server-verified writes – the system is subject to an attack by a revoked writer colluding with the server to make valid modifications to data. The only way to prevent this would be to broadcast the changed key to all users aggressively . Otherwise, a revoked writer will always be able to create data that looks valid and cheat (unsuspecting) readers who have not updated their key.

From the above discussion, it should be clear that lazy revocation is always susceptible to attacks mounted by revoked users in collusion with the server, unless a third (trusted) party is involved in each read and write access.

Finally, the server could mount the following attack, which we consider very difficult for the system to handle. In a *forking attack* [31], a server forks the state of a file between users. That is, the server separately maintains file updates for the users. The forked users never see each other's changes, and each user believes its state reflects reality. A higher level Byzantine agreement protocol, which is potentially expensive, might be necessary to address this issue [11]. Recently Mazières and Shasha [32] introduced the notion of *fork consistency* and a protocol to achieve it. Though their scheme does not prevent a forking attack, it makes it easier to detect.

5 Protocols

We now summarize the steps involved in protocols for creating, reading and writing as well as revoking users. We would like to remark again that all the keys and to-

Users	Key freshness	Collusion	None	D	L	LD	T	TD	TL	TLD
revoked reader	old keys	alone	f	f	f	–	–	–	–	–
		w/ server	c,d,f	c,d,f	c,d,f	c,d,f	c,d,f	c,d	c,d,f	c,d,f
revoked writer	old keys	alone	c,b,d	c,b,d	c,b,d	c,b,d	–	–	–	–
		w/ server	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d	c,b,d
	updated keys	alone	n/a	n/a	d	d	n/a	n/a	–	–
		w/ server	n/a	n/a	d	d	n/a	n/a	d	d

Table 1: Attacks tabulated against what is changed following a revocation. The heading row presents different choices in the component that is changed following a revocation: the read/write verification token is changed (T), the file's lockbox is changed (L), or the file itself is re-encrypted with a new key (D). The entries in the table correspond to the most serious attack that can be mounted, the letter code corresponding to those described in the main text. "n/a" indicates an impossible combination – such as readers having updated keys but files not being re-encrypted or lockboxes not changed. A "–" is used to denote that no attack is possible.

kens in these protocols are exchanged between owners and readers/writers via a secure channel with a session key – for instance, mutual authentication based on passwords. However, file data is not encrypted over the wire, but only integrity-protected with the session key.

1. **Initialize filegroup:** To initialize a filegroup, a user generates a pair of dual keys (file-sign and file-verify keys) for signing and verifying the contents of files in the filegroup. The user also generates the symmetric file-lockbox key.

2. **Create file:** First, the owner selects a filegroup for the new file. If there is no appropriate filegroup the owner initializes one and uses the corresponding keys (file-sign, file-verify, and file-lockbox keys) for this file. The owner also generates a write token and sends it to the server so that the server can verify all writes to this file.

3. **Read file:** A reader first obtains the name of the owner and the filegroup of the file he wishes to access, possibly after browsing the file system. The reader then checks if the version of the keys she has cached is greater then the version of the keys used to encrypt the file (which is stored in the header), in which case she does a key rotation to get the right version key. Otherwise, the reader gets the latest version key from the owner after appropriate authentication (via a secure channel). The reader then fetches the encrypted blocks of the desired file from the server, opens the lockboxes with the file-lockbox key, retrieves file-block keys from the lockbox, and decrypts the individual blocks. The integrity of the root hash (of the Merkle tree) provided by the server is first verified by using the file-verify key. To verify the integrity of the data, this root hash is compared against the root hash obtained by recomputing the Merkle hash tree using the file blocks retrieved from the server.

4. **Write file:** The writer obtains the latest version file-lockbox key and file-sign key, possibly from the owner if it is not cached. The writer then generates the file-block keys, encrypts individual blocks of the file using the corresponding file-block keys, and stores the encrypted blocks with the lockbox in the server. The server uses the write-token, provided by the writer at this time, to authorize the write. The writer then sends the entire Merkle hash tree in the clear to the server; the hash tree includes the root hash, signed with the file-sign key.

5. **Revoke user:** To revoke a user from accessing files in a filegroup, the owner generates the next version of the file-sign, file-verify, and file-lockbox keys. The owner then labels all files in the filegroup as needing re-encryption. If the revoked user is a writer, the owner changes the write-tokens in all the files of the filegroup as well.

6 Implementation

Using the protocols and ideas discussed in Section 3, we have designed Plutus and developed a prototype using OpenAFS [37]. In this section, we describe the architecture and the prototype of Plutus in detail.

6.1 Architecture of Plutus

Figure 3 summarizes the different components of Plutus, and where (server side or client side) they are implemented. Both the server and the clients have a network component, which is used to protect the integrity of all messages. In our implementation we protect the integrity of packets in the AFS RPC using HMAC [4]. Additionally, some messages such as those initiating read and write requests are encrypted. A 3DES session key, exchanged as part of the RPC setup, is optionally used to encrypt

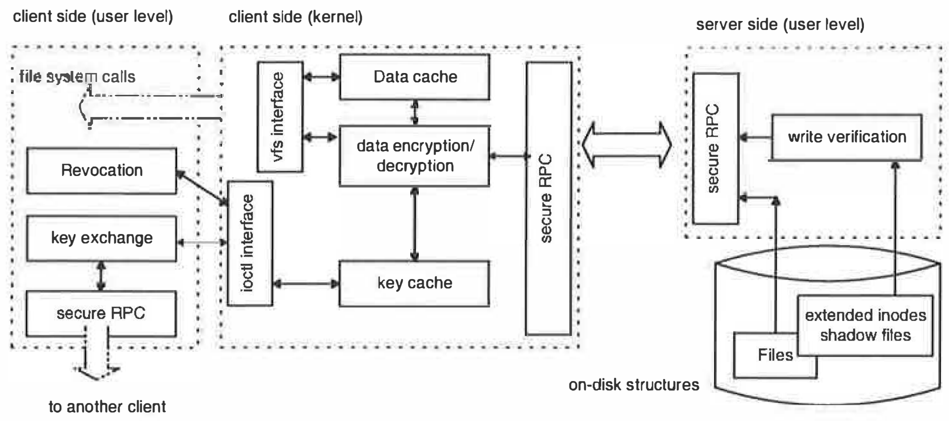


Figure 3: Architecture of Plutus.

these packets. The identities of all entities are established using 1024 bit RSA public/private keys.

The server has an additional component that validates writes. As described in Section 3.6, this component computes the SHA-1 hash of write tokens to authorize writes. This hashed token is passed on to the server, when the file is created, and is stored in the file's vnode (UNIX inode extension in AFS). Storing the token in the vnode instead of the directory simplifies write verification[2]. Owners change the stored token using a special ioctl call.

Most of the complexity of the implementation is at the client-side. We extended the OpenAFS kernel module by adding a key cache per user and a component to handle file data encryption and decryption. The key cache holds all keys used by the user, including file keys and identity keys (users' and servers' public keys). Currently all the encryptions and decryptions are done below the AFS cache; that is, we cache clear-text data. By doing this we encrypt (decrypt) file contents only when it is being transmitted to (received from) the server. The alternative of caching encrypted data would mean that each partial read/write would incur a block encryption/decryption, as would multiple reads/writes of the same block. We expect this to incur a substantial cryptographic overhead. Of course, caching unencrypted data opens up a security vulnerability on shared machines.

The other components of the client – revocation and key exchange – are implemented in user space. These components interact with the key cache through an extension to AFS's ioctl interface. The same client-server RPC interface is used for all inter-client communication.

Files are fragmented, and each fragment (blocks of size 4 KB) is encrypted independently with its own file-block (3DES) key. This 3DES key is kept in the fragment's lockbox together with the length of the fragment. The hashes of all the fragments are arranged in a Merkle hash tree, and the root signed (1024 bit RSA) with the file-sign-key. The leaves of the tree contain the lockbox of the corresponding fragment. The tree is kept in a "shadow file," on the server, and is shipped to the client, when the corresponding file is opened. On the client side, when blocks are updated, the respective new hashes are spliced into the tree. Then, the root hash is recomputed and signed when the cache is to be flushed to the server. At this time, the new tree is also sent back to the server.

6.2 Prototype

In building the Plutus prototype, we have made some modifications to the protocols to accommodate nuances of AFS. However, these modifications have little impact on the actual evaluation reported in the next section. For instance, currently AFS's RPC supports only authentication of the client by the server through a three step procedure. Recall that in Plutus design, the server never needs to authenticate a client. We use only the last two steps of this interface to achieve reverse authentication (i.e., client authenticating server) and session key exchange. To do this we need the server's public key, which can be succinctly implemented with self certifying pathnames [30], thus securely binding directories to servers.

The prototype uses a library that was built from the cryptographic routines in GnuPGP, with the following choice of primitives: 1024-bit RSA PKCS#1(version 1.5)[3] for public/private key encryption, SHA-1 for hashing and 3DES with CBC with Cipher Text Stealing [45] for file encryption.

[2]A similar problem was encountered in the context of storing inodes and small files together [16].

[3]For better security guarantees, RSA-OAEP is required; see Shoup's proposal [46] for more details.

7 Performance evaluation

In the preceding sections, we analyzed protocols of Plutus from a security perspective. We now evaluate Plutus from a performance perspective. In particular, we evaluate the design and the prototype of Plutus using (a) a trace from a running UNIX system, and (b) synthetic benchmarks. Using (a), the trace statistics, we argue the benefits of filegroups and the impact of lazy revocation. By measuring the overhead of Plutus using (b), synthetic benchmarks, we argue that though there is an overhead for the encryption/decryption, Plutus is quite practical; in fact, it compares favorably with SFS.

7.1 Trace evaluation

The trace that we use for evaluation is a 10-day complete file system trace (97.4 million requests, 129 GB data moved and 24 file systems) of a medium-sized workgroup using a 4-way HP-UX time-sharing server attached to several disk arrays and a total of 500 GB of storage space. This represents access to both NFS filesystems that the server exported, and accesses to local storage at the server. The trace was collected by instrumenting the kernel to log all file system calls at the syscall interface. Since this is above the file buffer cache, the numbers shown will be pessimistic to any system that attempts to optimize key usage on repeated access.

Key sharing	System		User	
	mean	max	mean	max
key/file	1,700	9,200	900	41,100
key/filegroup	11	57	6	23

Table 2: Using filegroups to aggregate keys.

Keys and filegroups

Table 2 presents the number of keys distributed among users. We classified all the user-ids in the system into System (such as root, bin, etc.) and User (regular users). The first row represents the number of keys that need to be distributed if a different key is used for each file in the system; the second row represents the number of keys distributed if filegroups are used. In this evaluation, we used the (mode bits, owner, group) tuple to define filegroups. The table presents numbers for both the maximum number of keys distributed by any user, and the mean number of keys distributed (averaged across all users who distributed at least one key). The table demonstrates the benefit of using filegroups clearly: the maximum number of keys distributed is reduced to 23, which is easy to manage. Note that even this is a pessimistic evaluation as it assumed a *cold key cache*.

Overhead of revocation

Table 3 presents parameters of the traced system that affect the overhead of performing a revocation. In this context we focus on the case where the owner of a filegroup wants to revoke another user's permission to read/write files in the owner's filegroup. We use these parameters to evaluate the overhead of performing a revocation, both in terms of carrying out the operations immediately following a revocation, and re-distributing the updated keys to other users. In the case of revoking a reader, the time spent immediately following a revocation is the time required to mark all files in the filegroup as "to be re-encrypted." In the case of revoking a writer this is the time to change the write verification key of all the files in the filegroup. For the system we traced, if a user revokes another user, this would involve marking 4,800 files to be re-encrypted, on average, and about 119,000 files, maximum. When a user (reader or writer) is revoked, other users (readers/writers) need to be given the updated key. Our evaluation shows that this number is typically very small: 2 on average and at most 11 in the worst case.

7.2 Cryptographic cost

Table 4 presents the impact of encryption/decryption on read and write latency. These are measurements of the cryptographic cost that includes write verification, data encryption, and wire-transmission overheads. These were done using code from Plutus' cryptography library on a 1.26 GHz Pentium 4 with 512 MB memory. In this evaluation, we used 4 KB as the size of the file fragment (corresponding to that of the prototype). As in the prototype, for data encryption, we used 1,024-bit RSA with a 256-bit file-verify key for reading and a 1,019-bit file-sign key for writing and 3DES CBC/CTS file-block key for bulk data encryption.

Owners incur a high one time cost to generate the read/write key pair; this is another reason why aggregating keys for multiple files using filegroups is beneficial. Though the write verification latency is negligible for writers and owners, if we choose to hide the identities of filegroups, then we pay an additional cost of decrypting it. The time spent in transmitting the Merkle hash tree depends on the size of the file being transmitted. In Plutus, block hashes are computed over 4 KB blocks, which contribute to about 1% overhead in data transmission.

For large files, the block encryption/decryption time dominates the cost of writing/reading the entire file. Though Plutus currently uses 3DES as the block cipher, from Dai's comparison of AES and 3DES [9], we expect a 3X speedup if AES were used.

Parameters	User			System		
	Highest	Second	Mean	Highest	Second	Mean
number of files	119,000	101,200	4,800	1,561,000	94,000	29,800
total bytes	17 GB	11 GB	0.6 GB	29 GB	14 GB	1.3 GB
number of readers	5	4	1.2	27	22	5.4
number of writers	6	5	0.7	15	14	1.7

Table 3: Parameters of the system that affect revocation. These are statistics indicating the number of files in a single filegroup owned by a user, the total size of all these files, the number of other users who have read permission to at least one of these files, and the number of other users who have write permission to at least one of these files. The number of readers and writers were determined by considering the accesses in the 10-day trace, while the static information was gathered by considering a snapshot of the filesystem taken at the end of the 10 days. The table separates statistics for regular users and system users.

File system operation	Crypto operation	Crypto cost	Incurred by	Frequency
Filegroup creation	RSA key generation	2500 ms	owner	per filegroup
File write	Block hash	0.11 ms	writer	per 4 KB block
	Block encrypt	0.59 ms	writer	per 4 KB block
	Merkle root sign	28.5 ms	writer	per file
	Write verify	0.01 ms	server	per file
File read	Block hash	0.11 ms	reader	per 4KB block
	Block decrypt	0.61 ms	reader	per 4 KB block
	Merkle root verify	8.5 ms	reader	per file
Wire integrity	Message encrypt	0.01 ms	all	per 100 byte message
	Message decrypt	0.01 ms	all	per 100 byte message
	Message hash	0.003 ms	all	per 100 byte message

Table 4: Cryptographic primitive cost. This table lists the cost of the basic cryptographic primitives, and the file systems operations where they are incurred. The root signature and verification is done only once per file read or write, irrespective of the size of the file. Wire integrity is needed only for messages, not for file contents.

7.3 Benchmark evaluation

We used a microbenchmark to compare the performance of Plutus to native OpenAFS and to SFS [30]. The microbenchmark we used is modeled on the Sprite LFS large file benchmarks. These involve reading and writing multiple 40 MB files with sequential and random accesses.

We used two identically configured machines, as in the previous section, connected with a Gigabit ethernet link. In all these experiments we restarted the client daemon, before reading/writing any file. We present the mean of 6 out of 10 runs, ignoring the top and bottom two outliers.

Table 5 presents the results of this evaluation. First, the table shows that the overhead of Plutus is primarily dependent on the choice of block cipher used. For instance, it takes 5.9s to decrypt 40MB with 3DES, which is about 75% of the average sequential read latency. Thus Plutus with no-crypto is faster than that with DES, which is in turn faster than with 3DES.

Second, Plutus performs as well as (if not better than) the other two encrypt-on-wire systems. In these comparisons it is important to compare systems that use block ciphers with similar security properties. In particular, the performance of Plutus with DES is slightly better than that of OpenAFS with fcrypt as the cipher: the fcrypt cipher is similar to DES. Though Plutus with 3DES is about 1.4 times slower than SFS, the latter uses ARC4, which is known to have a throughput about 14 times that of 3DES [9]. This leads us to believe that if Plutus were modified to use ARC4 or AES, it would compare well with SFS.

Note that this experiment is a pessimistic comparison between Plutus and the other two encrypt-on-wire systems. In the setting where there are several clients accessing data from the same server, Plutus would provide better server throughput because the server does not perform much crypto. This would translate to lower average latencies for Plutus.

8 Related work

Most file systems including those in MS Windows, traditional UNIX systems, and secure file systems [19, 22, 30] do not store files encrypted on the server. Of course, the user may decide to encrypt files before stor-

File systems	Crypto options	Read		Write	
		seq	rand	seq	rand
Plutus	w/ 3DES cipher	7.84 s	7.78 s	7.92 s	8.13 s
	w/ DES cipher	4.58 s	4.54 s	4.27 s	4.79 s
	w/o crypto	1.39 s	1.51 s	1.59 s	2.64 s
OpenAFS	w/o wire-crypto	1.28 s	1.31 s	1.57 s	1.67 s
	w/ wire-crypto	4.66 s	4.90 s	5.34 s	5.43 s
SFS	w/ crypto	5.55 s	5.30 s	4.47 s	7.21 s

Table 5: Performance of Plutus, OpenAFS (version 1.2.8) and SFS (version 0.7.2) accessing 40 MB files with random and sequential access. The crypto option for Plutus indicates the cipher used for block encryption; the OpenAFS crypto option indicates whether it uses wire-crypto or not; SFS uses wire-crypto. OpenAFS uses fcrypt [3] for block encryption whereas SFS uses ARC4 [23]. The version of Plutus w/o crypto still performed all the operations required to manage and maintain the Merkle hash tree; the results indicate that this overhead is small.

age but this overwhelms the user with the manual encryption/decryption and sharing the file with other users – while trying to minimize the amount of computation. This is precisely the problem that Plutus addresses.

Though MacOS X and Windows CIFS offer encrypted disks, they do not allow group sharing short of sharing a password.

8.1 Secure file systems

In encrypt-on-disk file systems, the clients encrypt all directories and their contents. The original work in this area is the Cryptographic File System (CFS) [5], which used a single key to encrypt an entire directory of files and depended on the underlying file system for authorization of writes. Later variants on this approach include TCFS [8], which uses a lockbox to protect only the keys, and Cryptfs [51]. Cepheus [13] uses group-managed lockboxes with a centralized key server and authorization at the trusted server. SNAD [35] also uses lockboxes and introduces several alternatives for verifying writes. The SiRiUS file system layers a cryptographic storage file system over heterogenous insecure storage such as NFS and Yahoo! Briefcase [21].

Encrypt-on-wire file systems protect the data from adversaries on the communication link. Hence all communication is protected, but the data is stored in plaintext. Systems that use encryption on the wire include NASD (Networked Attached Storage) [20], NFS over IPSec [24], SFS (Self-Certifying File System) [30], iSCSI [42], and OpenAFS with secure RPC.

In these systems the server is trusted with the data and meta-data. Even if users encrypt files, the server knows the filenames. This is not acceptable if the servers are untrustworthy, as in a distributed environment where servers can belong to multiple administrative domains. On the positive side, this simplifies space management because it is easy for the server to figure out the data blocks that are in use. A comprehensive evaluation of these systems appear in a previous study [40].

8.2 Untrusted servers

One way to recover from a malicious server corrupting the persistent store is to replicate the data on several servers. In the state machine approach [26, 44], clients read and write data to each replica. A client can recover a corrupted file by contacting enough replicas. The drawback to this method is that each replica must maintain a complete copy of the data.

Rabin's Information Dispersal Algorithm divides a file into several pieces, one for each replica [39]. While the aggregate space consumed by all the replicas is minimal, the system does not prevent or detect corruption.

Alon et al. describe a storage system resistant to corruption of data by half of the servers [2]. A client can recover from integrity-damaged files as long as a threshold number of servers remain uncorrupted.

9 Conclusion

This paper has introduced novel uses of cryptographic primitives applied to the problem of secure storage in the presence of untrusted servers and a desire for owner-managed key distribution. Eliminating almost all requirements for server trust (we still require servers not to destroy data – although we can detect if they do) and keeping key distribution (and therefore access control) in the hands of individual data owners provides a basis for a secure storage system that can protect and share data at very large scales and across trust boundaries.

The mechanisms described in this paper are used as building blocks to design Plutus, a comprehensive, secure, and efficient file system. We built a prototype implementation of this design by incorporating it into OpenAFS, and measured its performance on micro-benchmarks. We showed that the performance impact, due mostly to the cost of cryptography, is comparable to the cost of two popular schemes that encrypt on the wire. Yet, almost all of Plutus' cryptography is performed on clients, not servers, so Plutus has superior scalability along with stronger security.

Acknowledgements

We thank our shepherd Peter Honeyman for his comments and help while revising the paper. We thank Sameer Ajmani, Dan Boneh, Thomer Gil, Eu-Jin Goh, Dave Mazières, Chris Laas, Ethan Miller, Raj Rajagopalan, and Emil Sit for their helpful comments. We also thank Jeff Schiller, Garry Zacheiss, and Nickolai Zeldovich for producing logs of revocations on MIT's AFS protection servers.

References

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, pages 1–14, December 2002.

[2] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern. Scalable secure storage when half the system is faulty. In *ICALP*, 2000.

[3] T. Anderson. Specification of FCrypt: Encryption for AFS remote procedure calls,. http://www.transarc.ibm.com/~ota/fcrypt-paper.txt.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *CRYPTO*, pages 1–15, 1996.

[5] M. Blaze. A cryptographic file system for UNIX. In *CCS*, 1993.

[6] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46, 1999.

[7] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *OSDI*, pages 273–288, October 2000.

[8] G. Cattaneo, G. Persiano, A. Del Sorbo, A. Cozzolino, E. Mauriello, and R. Pisapia. Design and implementation of a transparent cryptographic file system for UNIX. Technical report, University of Salerno, 1997.

[9] W. Dai. Crypto++ version 4.2. http://www.eskimo.com/~weidai/cryptlib.html.

[10] C. Dalton and T. Choo. Trusted linux: An operating system approach. *CACM*, 44(2), February 2001.

[11] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *SRDS*, pages 4–13, 2001.

[12] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[13] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, MIT, June 1999.

[14] K. Fu, M. Kallahalla, S. Rajagopalan, and R. Swaminathan. Secure rotation on key sequences. Submitted for publication, 2002.

[15] K. Fu, M. Kallahalla, and R. Swaminathan. A simple protocol for maintaining fork consistency. Manuscript, 2002.

[16] G. Ganger and M. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Tech. Conf.*, pages 1–17, 1997.

[17] G. Ganger, P. Khosla, M. Bakkaloglu, M. Bigrigg, G. Goodson, S. Oguz, V. Pandurangan, C. Soules, J. Strunk, and J. Wylie. Survivable storage systems. In *DARPA Information Survivability Conference and Exposition, IEEE*, volume 2, pages 184–195, June 2001.

[18] D. Gifford, R. Needham, and M. Schroeder. The Cedar file system. In *CACM*, pages 288–298, March 1988.

[19] H. Gobioff, D. Nagle, and G. Gibson. Embedded security for network-attached storage. Technical Report CMU-CS-99-154, CMU, June 1999.

[20] H. Gobioff, D. Nagle, and G. A. Gibson. Integrity and performance in network attached storage. Technical Report CMU-CS-98-182, CMU, December 1998.

[21] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003. To appear.

[22] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM TOCS*, 6(1), February 1988.

[23] K. Kaukonen and R.Thayer. A stream cipher encryption algorithm "Arcfour". http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt.

[24] S. Kent and R. Atkinson. Security architecture for the Internet Protocol. Technical report, RFC 2401, November 1998.

[25] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, December 2000.

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[27] L. Lamport. Password authentication with insecure communication. *CACM*, 24(11):770-772, 1981.

[28] H. Lenstra. Divisors in residue classes. *Mathematics of computation*, pages 331–340, 1984.

[29] H. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[30] D. Mazières, M. Kaminsky, M. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, pages 124–139, December 1999.

[31] D. Mazières and D. Shasha. Don't trust your file server. In *HotOS*, pages 113–118, May 2001.

[32] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC*, pages 1–15, 2002.

[33] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of applied cryptography*. CRC press, 1997.

[34] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, volume 293, pages 369–378, 1987.

[35] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *FAST*, pages 1–13, January 2002.

[36] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, December 2002.

[37] OpenAFS. http://www.openafs.org/.

[38] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *FAST*, pages 89–102, January 2002.

[39] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *JACM*, 36(2):335–348, 1989.

[40] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evalauting storage system security. In *FAST*, pages 15–30, January 2002.

[41] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *SOSP*, pages 110–123, December 1999.

[42] J. Satran, D. Smith, K. Meth, C. Sapuntzakis, R. Haagens, E. Zeidner, P. Von Stamwitz, and L. Dalle Ore. iSCSI draft standard. Technical report, IETF, January 2002.

[43] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The ITC distributed file system: principles and design. In *SOSP*, pages 35–50, 1985.

[44] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[45] B. Schneier. *Applied Cryptography*. John Wiley & Sons, second edition, 1996.

[46] V. Shoup. A proposal for an ISO standard for public key encryption (version 2.1). http://www.shoup.net/iso_2_1.ps.Z.

[47] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: protecting data in compromised systems. In *OSDI*, October 2000.

[48] Trusted computing platform alliance (TCPA). http://www.trustedcomputing.org/.

[49] D. Tygar and M. Herlihy. How to make replicated data secure. In *CRYPTO*, pages 379–391, 1987.

[50] VMware GSX server. http://www.vmware.com/.

[51] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, University of California at Los Angeles, 1998.

Metadata Efficiency in Versioning File Systems

Craig A.N. Soules, Garth R. Goodson, John D. Strunk, Gregory R. Ganger
Carnegie Mellon University

Abstract

Versioning file systems retain earlier versions of modified files, allowing recovery from user mistakes or system corruption. Unfortunately, conventional versioning systems do not efficiently record large numbers of versions. In particular, versioned metadata can consume as much space as versioned data. This paper examines two space-efficient metadata structures for versioning file systems and describes their integration into the Comprehensive Versioning File System (CVFS), which keeps all versions of all files. *Journal-based metadata* encodes each metadata version into a single journal entry; CVFS uses this structure for inodes and indirect blocks, reducing the associated space requirements by 80%. *Multiversion b-trees* extend each entry's key with a timestamp and keep current and historical entries in a single tree; CVFS uses this structure for directories, reducing the associated space requirements by 99%. Similar space reductions are predicted via trace analysis for other versioning strategies (e.g., on-close versioning). Experiments with CVFS verify that its current-version performance is similar to that of non-versioning file systems while reducing overall space needed for history data by a factor of two. Although access to historical versions is slower than conventional versioning systems, checkpointing is shown to mitigate and bound this effect.

1 Introduction

Self-securing storage [41] is a new use for versioning in which storage servers internally retain file versions to provide detailed information for post-intrusion diagnosis and recovery of compromised client systems [40]. We envision self-securing storage servers that retain every version of every file, where every modification (e.g., a WRITE operation or an attribute change) creates a new version. Such *comprehensive versioning* maximizes the information available for post-intrusion diagnosis. Specifically, it avoids pruning away file versions, since this might obscure intruder actions. For self-securing storage, *pruning techniques* are particularly dangerous when they rely on client-provided information, such as CLOSE operations — the versioning is being done specifically to protect stored data from malicious clients.

Obviously, finite storage capacities will limit the duration of time over which comprehensive versioning is possible. To be effective for intrusion diagnosis and recovery, this duration must be greater than the intrusion detection latency (i.e., the time from an intrusion to when it is detected). We refer to the desired duration as the *detection window*. In practice, the duration is limited by the rate of data change and the space efficiency of the versioning system. The rate of data change is an inherent aspect of a given environment, and an analysis of several real environments suggests that detection windows of several weeks or more can be achieved with only a 20% cost in storage capacity [41].

In a previous paper [41], we described a prototype self-securing storage system. By using standard copy-on-write and a log-structured data organization, the prototype provided comprehensive versioning with minimal performance overhead ($<$10%) and reasonable space efficiency. In that work, we discovered that a key design requirement is efficient encoding of metadata versions (the additional information required to track the data versions). While copy-on-write reduces data versioning costs, conventional versioning implementations still involve one or more new metadata blocks per version. On average, the metadata versions require as much space as the versioned data, halving the achievable detection window. Even with less comprehensive versioning, such as Elephant [37] or VMS [29], the metadata history can become almost ($\approx$80%) as large as the data history.

This paper describes and evaluates two methods of storing metadata versions more compactly: journal-based metadata and multiversion b-trees. Journal-based metadata encodes each version of a file's metadata in a journal entry. Each entry describes the difference between two versions, allowing the system to roll-back to the earlier version of the metadata. Multiversion b-trees retain all versions of a metadata structure within a single tree. Each entry in the tree is marked with a timestamp indicating the time over which the entry is valid.

The two mechanisms have different strengths and weaknesses. We discuss these and describe how both techniques are integrated into a comprehensive versioning file system called CVFS. CVFS uses journal-based metadata for inodes and indirect blocks to encode changes to attributes and file data pointers; doing so reduces the space used for their histories by 80%. CVFS implements directories as multiversion b-trees to encode additions and removals of directory entries; doing so reduces the space used for their histories by 99%. Combined, these mechanisms nearly double the potential detection window over conventional versioning mechanisms, without increasing the access time to current versions of the data.

Journal-based metadata and multiversion b-trees are also valuable for conventional versioning systems. Using these mechanisms with on-close versioning and snapshots would provide similar reductions in versioned metadata. For on-close versioning, this reduces the total space required by nearly 35%, thereby reducing the pressure to prune version histories. Identifying solid heuristics for such pruning remains an open area of research [37], and less pruning means fewer opportunities to mistakenly prune important versions.

The rest of this paper is divided as follows. Section 2 discusses conventional versioning and motivates this work. Section 3 discusses the two space-efficient metadata versioning mechanisms and their tradeoffs. Section 4 describes the CVFS versioning file system. Section 5 analyzes the efficiency of CVFS in terms of space efficiency and performance. Section 6 describes how our versioning techniques could be applied to other systems. Section 7 discusses additional related work. Section 8 summarizes the paper's contributions.

2 Versioning and Space Efficiency

Every modification to a file inherently results in a new version of the file. Instead of replacing the previous version with the new one, a *versioning file system* retains both. Users of such a system can then access any historical versions that the system keeps as well as the most recent one. This section discusses uses of versioning, techniques for managing the associated capacity costs, and our goal of minimizing the metadata required to track file versions.

2.1 Uses of Versioning

File versioning offers several benefits to both users and system administrators. These benefits can be grouped into three categories: recovery from user mistakes, recovery from system corruption, and analysis of historical changes. Each category stresses different features of the versioning system beneath it.

Recovery from user mistakes: Human users make mistakes, such as deleting or erroneously modifying files. Versioning can help [17, 29, 37]. Recovery from such mistakes usually starts with some a priori knowledge about the nature of the mistake. Often, the exact file that should be recovered is known. Additionally, there are only certain versions that are of any value to the user; intermediate versions that contain incomplete data are useless. Therefore, versioning aimed at recovery from user mistakes should focus on retaining key versions of important files.

Recovery from system corruption: When a system becomes corrupted, administrators generally have no knowledge about the scope of the damage. Because of this, they restore the entire state of the file system from some well-known "good" time. A common versioning technique to help with this is the online *snapshot*. Like a backup, a snapshot contains a version of every file in the system at a particular time. Thus, snapshot systems present sets of known-valid system images at a set of well-known times.

Analysis of historical changes: A history of versions can help answer questions about how a file reached a certain state. For example, version control systems (e.g., RCS [43], CVS [16]) keep a complete record of committed changes to specific files. In addition to selective recovery, this record allows developers to figure out who made specific changes and when those changes were made. Similarly, self-securing storage seeks to enable post-intrusion diagnosis by providing a record of what happened to stored files before, during, and after an intrusion. We believe that every version of every file should be kept. Otherwise, intruders who learn the pruning heuristic will leverage this information to prune any file versions that might disclose their activities. For example, intruders may make changes and then quickly revert them once damage is caused in order to hide their tracks. With a complete history, administrators can determine which files were changed and estimate damage. Further, they can answer (or at least construct informed hypotheses for) questions such as "When and how did the intruder get in?" and "What was their goal?" [40].

2.2 Pruning Heuristics

A true comprehensive versioning system keeps all versions of all files for all time. Such a system could support all three goals described above. Unfortunately, storing this much information is not practical. As a result, all versioning systems use *pruning heuristics*. These pruning heuristics determine when versions should be created and when they should be removed. In other words, pruning heuristics determine which versions to keep from the total set of versions that would be available in a comprehensive versioning system.

2.2.1 Common heuristics

A common pruning technique in versioning file systems is *on-close* versioning. This technique keeps only the last version of a file from each session; that is, each CLOSE of a file creates a distinct version. For example, the VMS file system [29] retains a fixed number of versions for each file. VMS's pruning heuristic creates a version after each CLOSE of a file and, if the file already has the maximum number of versions, removes the oldest remaining version of the file. The more recent Elephant file system [37] also creates new versions after each CLOSE; however, it makes additional pruning decisions based on a set of rules derived from observed user behavior.

Version control systems prune in two ways. First, they retain only those versions explicitly committed by a user. Second, they retain versions for only an explicitly-chosen subset of the files on a system.

By design, snapshot systems like WAFL [19] and Venti/Plan9 [34] prune all of the versions of files that are made between snapshots. Generally, these systems only create and delete snapshots on request, meaning that the system's administrator decides most aspects of the pruning heuristic.

2.2.2 Information Loss

Pruning heuristics act as a form of lossy compression. Rather than storing every version of a file, these heuristics throw some data away to save space. The result is that, just as a JPEG file loses some of its visual clarity with lossy compression, pruning heuristics reduce the clarity of the actions that were performed on the file.

Although this loss of information could result in annoyances for users and administrators attempting to recover data, the real problem arises when versioning is used to analyze historical changes. When versioning for intrusion survival, as in the case of self-securing storage, pruning heuristics create holes in the administrator's view of the system. Even creating a version on every CLOSE is not enough, as malicious users can leverage this heuristic to hide their actions (e.g., storing exploit tools in an open file and then truncating the file to zero before closing it).

To avoid traditional pruning heuristics, self-securing storage employs comprehensive versioning over a fixed window of time, expiring versions once they become older than the given window. This detection window can be thought of as the amount of time that an administrator has to detect, diagnose, and recover from an intrusion. As long as an intrusion is detected within the window, the administrator has access to the entire sequence of modifications since the intrusion.

2.3 Lossless Version Compression

For a system to avoid pruning heuristics, even over a fixed window of time, it needs some form of lossless version compression. Lossless version compression can also be combined with pruning heuristics to provide further space reductions in conventional systems. To maximize the benefits, a system must attempt to compress both versioned data and versioned metadata.

Data: Data block sharing is a common form of lossless compression in versioning systems. Unchanged data blocks are shared between versions by having their individual metadata point to the same physical block. Copy-on-write is used to avoid corrupting old versions if the block is modified.

An improvement on block sharing is byte-range differencing between versions. Rather than keeping the data blocks that have changed, the system keeps the bytes that have changed [27]. This is especially useful in situations where a small change is made to the file. For example, if a single byte is inserted at the beginning of a file, a block sharing system keeps two full copies of the entire file (since the data of every block in the file is shifted forward by one byte); for the same scenario, a differencing system only stores the single byte that was added and a small description of the change.

Another recent improvement in data compression is hash-based data storage [31, 34]. These methods recognize identical blocks or ranges of data across the system and store only one copy of the data. This method is quite effective for snapshot versioning systems, and could likely be applied to other versioning systems with similar results.

Metadata: Conventional versioning file systems keep a full copy of the file metadata with each version. While it simplifies version access, this method quickly exhausts capacity, since even small changes to file data or attributes result in a new copy of the metadata.

Figure 1 shows an example of how the space overhead of versioned metadata can become a problem in a conventional versioning system. In this example, a program is writing small log entries to the end of a large file. Since several log entries fit within a single data block, appending entries to the end of the file produces several different versions of the same block. Because each versioned data block has a different location on disk, the system must create a new version of the indirect block to track its location. In addition, the system must write a new version of the inode to track the location of the versioned indirect block. Since any data or metadata change will always result in a new version of the inode, each version is tracked using a pointer to that version's inode. Thus, writing a

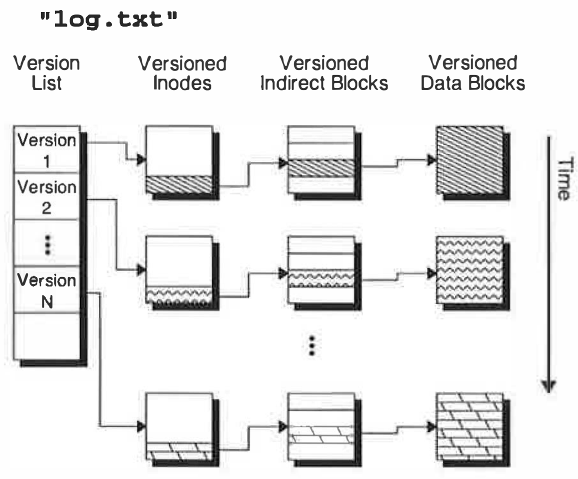


Figure 1: **Conventional versioning system.** In this example, a single logical block of file "log.txt" is overwritten several times. With each new version of the data block, new versions of the indirect block and inode that reference it are created. Notice that although only a single pointer has changed in both the indirect block and the inode, they must be rewritten entirely, since they require new versions. The system tracks each version with a pointer to that version's inode.

single data block results in a new indirect block, a new inode, and an entry in the version list, resulting in more metadata being written than data.

Access patterns that create such metadata versioning problems are common. Many applications create or modify files piece by piece. In addition, distributed file systems such as NFS create this behavior by breaking large updates of a file into separate, block-sized updates. Since there is no way for the server to determine if these block-sized writes are one large update or several small ones, each must be treated as a separate update, resulting in several new versions of the file.

Again, the solution to this problem is some form of differencing between the versions. Mechanisms for creating and storing differences of metadata versions are the main focus of this work.

2.4 Objective

In a perfect world we could keep all versions of all files for an infinite amount of time with no impact on performance. This is obviously not possible. The objective of this work is to minimize the space overhead of versioned metadata. For self-securing storage, doing so will increase the detection window. For other versioning purposes, doing so will reduce the pressure to prune. Because this space reduction will require compressing metadata versions, it is also important that the performance overhead of both version creation and version access be minimized.

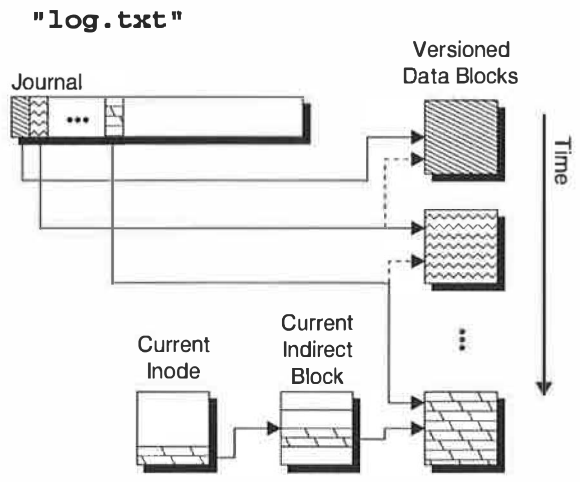


Figure 2: **Journal-based metadata system.** Just as in Figure 1, this figure shows a single logical block of file "log.txt" being overwritten several times. Journal-based metadata retains all versions of the data block by recording each in a journal entry. Each entry points to both the new block and the block that was overwritten. Only the current version of the inode and indirect block are kept, significantly reducing the amount of space required for metadata.

3 Efficient Metadata Versioning

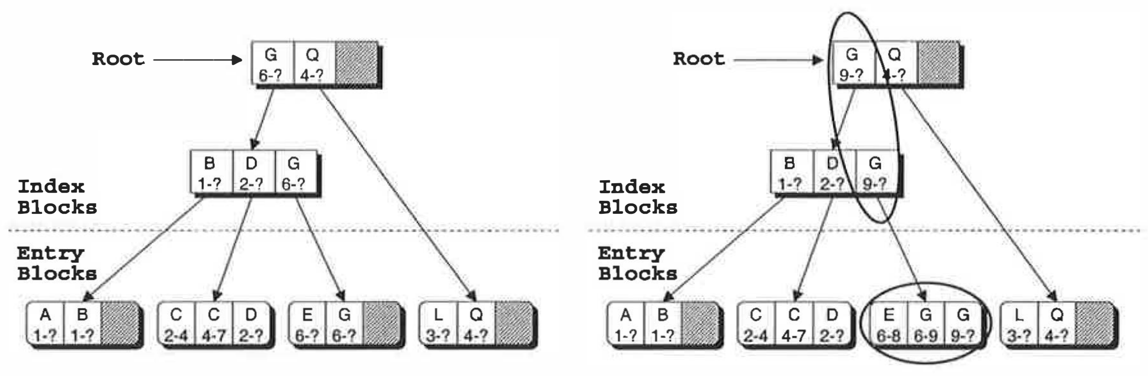
One characteristic of versioned metadata is that the actual changes to the metadata between versions are generally quite small. In Figure 1, although an inode and an indirect block are written with each new version of the file, the only change to the metadata is an update to a single block pointer. The system can leverage these small changes to provide much more space-efficient metadata versioning. This section describes two methods that leverage small metadata modifications, and Section 4 describes an implementation of these solutions.

3.1 Journal-based Metadata

Journal-based metadata maintains a full copy of the current version's metadata and a journal of each previous metadata change. To recreate old versions of the metadata, each change is undone backward through the journal until the desired version is recreated. This process of undoing metadata changes is referred to as *journal rollback*.

Figure 2 illustrates how journal-based metadata works in the example of writing log entries. Just as in Figure 1, the system writes a new data block for each version; however, in journal-based metadata, these blocks are tracked using small journal entries that track the locations of the new and old blocks. By keeping the current version of the metadata up-to-date, the journal entries can be rolled-back to any previous version of the file.

In addition to storing version information, the journal can



(a) Initial tree structure.

(b) After removal of E and update of G.

Figure 3: **Multiversion b-tree.** This figure shows the layout of a multiversion b-tree. Each entry of the tree is designated by a ⟨user-key, timestamp⟩ tuple which acts as a key for the entry. A question mark (?) in the timestamp indicates that the entry is valid through the current time. Different versions of an entry are separate entries using the same user-kcy with different timestamps. Entries are packed into entry blocks, which are tracked using index blocks. Each index pointer holds the key of the last entry along the subtree that it points to.

be used as a write-ahead log for metadata consistency, just as in a conventional journaling file system. To do so, the new block pointer must be recorded in addition to the old. Using this, a journal-based metadata implementation can safely maintain the current version of the metadata in memory, flushing it to disk only when it is forced from the cache.

3.1.1 Space vs. Performance

Journal-based metadata is more space efficient than conventional versioning. However, it must pay a performance penalty for recreating old versions of the metadata. Since each entry written between the current version and the requested version must be read and rolled-back, there is a linear relation between the number of changes to a file and the performance penalty for recreating old versions.

One way the system can reduce this overhead is to *checkpoint* a full copy of a file's metadata to the disk occasionally. By storing checkpoints and remembering their locations, a system can start journal roll-back from the closest checkpoint in time rather than always starting with the current version. The frequency with which these checkpoints are written dictates the space/performance trade-off. If the system keeps a checkpoint with each modification, journal-based metadata performs like a conventional versioning scheme (using the most space, but offering the best back-in-time performance). However, if no checkpoints are written, the only full instance of the metadata is the current version, resulting in the lowest space utilization but reduced back-in-time performance.

3.2 Multiversion B-trees

A multiversion b-tree is a variation on standard b-trees that keeps old versions of entries in the tree [2]. As in a standard b-tree, an entry in a multiversion b-tree contains a key/data pair; however, the key consists of both a user-defined key and the time at which the entry was written. With the addition of this time-stamp, the key for each version of an entry becomes unique. Having unique keys means that entries within the tree are never overwritten; therefore, multiversion b-trees can have the same basic structure and operations as a standard b-tree. To facilitate current version lookups, entries are sorted first by the user-defined key and then by the timestamp.

Figure 3a shows an example of a multiversion b-tree. Each entry contains both the user-defined key and the time over which the entry is valid. The entries are packed into entry blocks, which act as the leaf nodes of the tree. The entry blocks are tracked using index blocks, just as in standard b+trees. In this example, each pointer in the index block references the last entry of the subtree beneath it. So in the case of the root block, the G subtree holds all entries with values less than or equal to G, with $\langle G, 6-?\rangle$ as its last entry. The Q subtree holds all entries with values between G and Q, with $\langle Q, 4-?\rangle$ as its last entry.

Figure 3b shows the tree after a remove of entry E and an update to entry G. When entry E is removed at time 8, the only change is an update to the entry's timestamp. This indicates that E is only valid from time 6 through time 8. When entry G is updated at time 9, a new entry is created and associated with the new data. Also, the old entry for G must be updated to indicate its bounded window of validity. In this case, the index blocks must also be updated to reflect the new state of the subtree, since the last entry of the subtree has changed.

Since both current and history entries are stored in the same tree, accesses to old and current versions have the same performance. For this reason, large numbers of history entries can decrease the performance of accessing current entries.

3.3 Solution Comparison

Both journal-based metadata and multiversion b-trees reduce the space utilization of versioning but incur some performance penalty. Journal-based metadata pays with reduced back-in-time performance. Multiversion b-trees pay with reduced current version performance.

Because the two mechanisms have different drawbacks, they each perform certain operations more efficiently. As mentioned above, the number of history entries in a multiversion b-tree can adversely affect the performance of accessing the current version. This emerges in two situations: linear scan operations and files with a large number of versions. The penalty on lookup operations is reduced by the logarithmic nature of the tree structure, but large numbers of history entries can increase tree depth. Linear scanning of all current entries requires accessing every entry in the tree, which becomes expensive if the number of history entries is high. In both of these cases, it is better to use journal-based metadata.

When lookup of a single entry is common or history access time is important, it is preferable to use multiversion b-trees. Using a multiversion b-tree, all versions of the entry are located together in the tree and have logarithmic lookup time (for both current and history entries), giving a performance benefit over the linear roll-back operation required by journal-based metadata.

4 Implementation

We have integrated journal-based metadata and multiversion b-trees into a comprehensive versioning file system, called CVFS. CVFS provides comprehensive versioning within our self-securing NFS server prototype. Because of this, some of our design decisions (such as the implementation of a strict detection window) are specific to self-securing storage. Regardless, these structures would be effective in most versioning systems.

4.1 Overview

Since current versions of file data must not be overwritten in a comprehensive versioning system, CVFS uses a log-structured data layout similar to LFS [36]. Not only does this eliminate overwriting of old versions on disk, but it also improves update performance by combining data and metadata updates into a single disk write.

CVFS uses both mechanisms described in Section 3. It uses journal-based metadata to version file data pointers and file attributes, and multiversion b-trees to version directory entries. We chose this division of methods based on the expected usage patterns of each. Assuming many versions of file attributes and a need to access them in their entirety most of the time, we decided that journal-based metadata would be more efficient. Directories, on the other hand, are updated less frequently than file metadata and a large fraction of operations are entry lookup rather than full listing. Thus, the cost of having history entries within the tree is expected to be lower.

Since the only pruning heuristic in CVFS is expiration, it requires a cleaner to find and remove expired versions. Although CVFS's background cleaner is not described in detail here, its implementation closely resembles the cleaner in LFS. The only added complication is that, when moving a data block in a versioning system, the cleaner must update all of the metadata versions that point to the block. Locating and modifying all of this metadata can be expensive. To address this problem, each data block on the disk is assigned a virtual block number. This allows us to move the physical location of the data and only have to update a single pointer within a virtual indirection table, rather than all of the associated metadata.

4.2 Layout and Allocation

Because of CVFS's log-structured format, disk space is managed in contiguous sets of disk blocks called *segments*. At any particular time, there is a single *write segment*. All data block allocations are done within this segment. Once the segment is completely allocated, a new write segment is chosen. Free segments on the disk are tracked using a bitmap.

As CVFS performs allocations from the write segment, the allocated blocks are marked as either journal blocks or data blocks. Journal blocks hold journal entries, and they contain pointers that string all of the journal blocks together into a single contiguous journal. Data blocks contain file data or metadata checkpoints.

CVFS uses inodes to store a file's metadata, including file size, access permissions, creation time, modification time, and the time of the oldest version still stored on the disk. The inode also holds direct and indirect data pointers for the associated file or directory. CVFS tracks inodes with a unique inode number. This inode number indexes into a table of inode pointers that are kept at a

Entry Type	Description	Cause
Attribute	Holds new inode attribute information	Inode change
Delete	Holds inode number and delete time	Inode change
Truncate	Holds the new size of the file	File data change
Write	Points to the new file data	File data change
Checkpoint	Points to checkpointed metadata	Metadata checkpoint / Inode change

Table 1: **Journal entry types.** This table lists the five types of journal entry. Journal entries are written when inodes are modified, file data is modified, or file metadata is flushed from the cache.

fixed location on the disk. Each pointer holds the block number of the most current metadata checkpoint for that file, which is guaranteed to hold the most current version of the file's inode. The in-memory copy of an inode is always kept up-to-date with the current version, allowing quick access for standard operations. To ensure that the current version can always be accessed directly off the disk, CVFS checkpoints the inode to disk on a cache flush.

4.3 The Journal

The string of journal blocks that runs through the segments of the disk is called the *journal*. Each journal block holds several time-ordered, variably-sized journal entries. CVFS uses the journal to implement both conventional file system journaling (a.k.a. write-ahead logging) and journal-based metadata.

Each journal entry contains information specific to a single change to a particular file. This information must be enough to do both roll-forward and roll-back of the metadata. Roll-forward is needed for update consistency in the face of failures. Roll-back is needed to reconstruct old versions. Each entry also contains the time at which the entry was written and a pointer to the location of the previous entry that applies to this particular file. This pointer allows us to trace the changes of a single file through time.

Table 1 lists the five different types of journal entries. CVFS writes entries in three different cases: inode modifications (creation, deletion, and attribute updates), data modifications (writing or truncating file data), and metadata checkpoints (due to a cache flush or history optimization).

4.4 Metadata

There are three types of file metadata that can be altered individually: inode attributes, file data pointers, and directory entries. Each has characteristics that match it to a particular method of metadata versioning.

4.4.1 Inode Attributes

There are four operations that act upon inode attributes: creation, deletion, attribute updates, and attribute lookups.

CVFS creates inodes by building an initial copy of the new inode and checkpointing it to the disk. Once this checkpoint completes and the inode pointer is updated, the file is accessible. The initial checkpoint entry is required because the inode cannot be read through the inode pointer table until a checkpoint occurs. CVFS's default checkpointing policy bounds the back-in-time access performance to approximately 150ms as is described in Section 5.3.2.

To delete an inode, CVFS writes a "delete" journal entry, which notes the inode number of the file being deleted. A flag is also set in the current version of the inode, specifying that the file was deleted, since the deleted inode cannot actually be removed from the disk until it expires.

CVFS stores attribute modifications entirely within a journal entry. This journal entry contains the value of the changed inode attributes both before and after the modification. Therefore, an attribute update involves writing a single journal entry, and updating the current version of the inode in memory.

CVFS accesses the current version of the attributes by reading in the current inode, since all of the attributes are stored within it. To access old versions of the attributes, CVFS traverses the journal entries searching for modifications that affect the attributes of that particular inode. Once roll-back is complete, the system is left with a copy of the attributes at the requested point in time.

4.4.2 File Data Pointers

CVFS tracks file data locations using direct and indirect pointers [30]. Each file's inode contains thirty direct pointers, as well as one single, one double and one triple indirect pointer.

When CVFS writes to a file, it allocates space for the new data within the current write segment and creates a "write" journal entry. The journal entry contains pointers to the data blocks within the segment, the range of logical block numbers that the data covers, the old size of the file, and pointers to the old data blocks that were overwritten (if there were any). Once the journal entry is allocated, CVFS updates the current version of the metadata to point at the new data.

If a write is larger than the amount of data that will fit within the current write segment, CVFS breaks the write into several data/journal entry pairs across different segments. This compartmentalization simplifies cleaning.

To truncate a file, CVFS first checkpoints the file to the log. This is necessary because CVFS must be able to locate truncated indirect blocks when reading back-in-time. If they are not checkpointed, then the information in them will be lost during the truncate; although earlier journal entries could be used to recreate this information, such entries could expire and leave the detection window, resulting in lost information. Once the checkpoint is complete, a "truncate" journal entry is created containing both a pointer to the checkpointed metadata and the new size of the file.

To access current file data, CVFS finds the most current inode and reads the data pointers directly, since they are guaranteed to be up-to-date. To access historical data versions, CVFS uses a combination of checkpoint tracking and journal roll-back to recreate the desired version of the requested data pointers. CVFS's checkpoint tracking and journal roll-back work together in the following way. Assume a user wishes to read data from a file at time T. First, CVFS locates the oldest checkpoint it is tracking with time T_c such that $T_c \geq T$. Next, it searches backward from that checkpoint through the journal looking for changes to the block numbers being read. If it finds an older version of a block that applies, it will use that. Otherwise, it reads the block from the checkpointed metadata.

To illustrate this journal rollback, Figure 4 shows a sequence of updates to block 3 of inode 4 interspersed with checkpoints of inode 4. Each block update and inode checkpoint is labeled with the time t that it was written. To read block 3 at time $T_1 = 12$, CVFS first reads the checkpoint at time $t = 18$, then reads journal entries to see if a different data block should be used. In this case, it finds that the block was overwritten at time $t = 15$, and so returns the older block written at time $t = 10$. In the case of time $T_2 = 5$, CVFS starts with the checkpoint at time $t = 7$, and then reads the journal entry, and realizes that no such block existed at time $t = 5$.

4.4.3 Directory Entries

Each directory in CVFS is implemented as a multiversion b-tree. Each entry in the tree represents a directory entry; therefore, each b-tree entry must contain the entry's name, the inode number of the associated file, and the time over which the entry is valid. Each entry also contains a fixed-size hash of the name. Although the actual name must be used as the key while searching through the entry blocks, this fixed-size hash allows the index blocks to use space-efficient fixed-size keys.

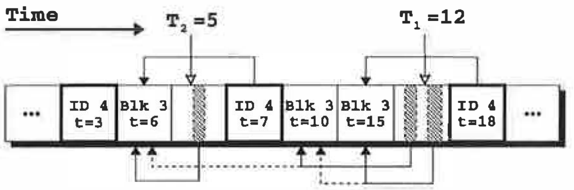


Figure 4: **Back-in-time access.** This diagram shows a series of checkpoints of inode 4 (highlighted with a dark border) and updates to block 3 of inode 4. Each checkpoint and update is marked with a time t at which the event occured. Each checkpoint holds a pointer to the block that is valid at the time of the checkpoint. Each update is accompanied by a journal entry (marked by thin, grey boxes) which holds a pointer to the new block (solid arrow) and the old block that it overwrote (dashed arrow, if one exists).

CVFS uses a full data block for each entry block of the tree, and sorts the entries within it first by hash and then by time. Index nodes of the tree are also full data blocks consisting of a set of index pointers also sorted by hash and then by time. Each index pointer is a ⟨subtree, hash, time-range⟩ tuple, where *subtree* is a pointer to the appropriate child block, *hash* is the name hash of the last entry along the subtree, and *time-range* is the time over which that same entry is valid.

With this structure, lookup and listing operations on the directory are the same as with a standard b-tree, except that the requested time of the operation becomes part of the key. For example, in Figure 3a, a lookup of $\langle C,\ 5\rangle$ searches through the tree for entries with name C, and then checks the time-ranges of each to determine the correct entry to return (in this case $\langle C,\ 4-7\rangle$). A listing of the directory at time 5 would do an in-order tree traversal (just as in a standard b-tree), but would exclude any entries that are not valid at time 5.

Insert, remove, and update are also very similar. Insert is identical, with the time-range of the new entry starting at the current time. Remove is an update of the time-range for the requested name. For example, in Figure 3b, entry E is removed at time 8. Update is an atomic remove and insert of the same entry name. For example, in Figure 3b, entry G is updated at time 9. This involves atomically removing the old entry G at time 9 (updating the time-range), and inserting entry G at time 9 (the new entry $\langle G,\ 9-?\rangle$).

Labyrinth Traces					
		Versioned Data	**Versioned Metadata**	**Metadata Savings**	**Total Savings**
Files:	Conventional versioning	123.4 GB	142.4 GB		
	Journal-based metadata	123.4 GB	4.2 GB	97.1%	52.0%
Directories:	Conventional versioning	—	9.7 GB		
	Multiversion b-trees	—	0.044 GB	99.6%	99.6%
Total:	Conventional versioning	123.4 GB	152.1 GB		
	CVFS	123.4 GB	4.244 GB	**97.2%**	**53.7%**

Lair Traces					
		Versioned Data	**Versioned Metadata**	**Metadata Savings**	**Total Savings**
Files:	Conventional versioning	74.5 GB	34.8 GB		
	Journal-based metadata	74.5 GB	1.1 GB	96.8%	30.8%
Directories:	Conventional versioning	—	1.8 GB		
	Multiversion b-trees	—	0.0064 GB	99.7%	99.7%
Total:	Conventional versioning	74.5 GB	152.1 GB		
	CVFS	74.5 GB	1.1064 GB	**97.0%**	**32.0%**

Table 2: **Space utilization.** This table compares the space utilization of conventional versioning with CVFS, which uses journal-based metadata and multiversion b-trees. The space utilization for versioned data is identical for conventional versioning and journal-based metadata because neither address data beyond block sharing. Directories contain no versioned data because they are entirely a metadata construct.

5 Evaluation

The objective of this work is to reduce the space overheads of versioning without reducing the performance of current version access. Therefore, our evaluation of CVFS is done in two parts. First, we analyze the space utilization of CVFS. We find that using journal-based metadata and multiversion b-trees reduces space overhead for versioned metadata by over 80%. Second, we analyze the performance characteristics of CVFS. We find that it performs similarly to non-versioning systems for current version access, and that back-in-time performance can be bounded to acceptable levels.

5.1 Experimental Setup

For the evaluation, we used CVFS as the underlying file system for S4, our self-securing NFS server. S4 is a user-level NFS server written for Linux that uses the SCSI-generic interface to directly access the disk. S4 exports an NFSv2 interface and treats it as a security perimeter between the storage system and the client operating system. Although the NFSv2 specification requires that all changes be synchronous, S4 also has an asynchronous mode of operation, allowing us to more thoroughly analyze the performance overheads of our metadata versioning techniques.

In all experiments, the client system has a 550 MHz Pentium III, 128 MB RAM, and a 3Com 3C905B 100 Mb network adapter. The servers have two 700 MHz Pentium IIIs, 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro100 100 Mb network adapter. The client and server are on the same 100 Mb network switch.

5.2 Space Utilization

We used two traces, labelled *Labyrinth* and *Lair*, to evaluate the space utilization of our system. The *Labyrinth* trace is from an NFS server at Carnegie Mellon holding the home directories and CVS repository that support the activities of approximately 30 graduate students and faculty; it records approximately 164 GB of data traffic to the NFS server over a one-month period. The *Lair* trace [13] is from a similar environment at Harvard; it records approximately 103 GB of data traffic over a one-week period. Both were captured via passive network monitoring.

We replayed each trace onto both a standard configuration of CVFS and a modified version of CVFS. The modified version simulates a conventional versioning system by checkpointing the metadata with each modification. It also performs copy-on-write of directory blocks, overwriting the entries in the new blocks (that is, it uses normal b-trees). By observing the amount of allocated data for each request, we calculated the exact overheads of our two metadata versioning schemes as compared to a conventional system.

Table 2 compares the space utilization of versioned files for the two traces using conventional versioning and journal-based metadata. There are two space overheads for file versioning: versioned data and versioned metadata. The overhead of versioned data is the overwritten or deleted data blocks that are retained. In both

Labyrinth Traces							
		Versioned Data	Versioned File Metadata	Versioned Directories	Version Ratio	Metadata Savings	Total Savings
Comprehensive:	Conventional	123.4 GB	142.4 GB	9.7 GB			
	CVFS	123.4 GB	4.2 GB	0.044 GB	1:1	97%	54%
On close():	Conventional	55.3 GB	30.6 GB	2.4 GB			
	CVFS	55.3 GB	2.1 GB	0.012 GB	1:2.8	94%	35%
6 minute Snapshots:	Conventional	53.2 GB	11.0 GB	2.4 GB			
	CVFS	53.2 GB	1.3 GB	0.012 GB	1:11.7	90%	18%
1 hour Snapshots:	Conventional	49.7 GB	5.1 GB	2.4 GB			
	CVFS	49.7 GB	0.74 GB	0.012 GB	1:20.8	90%	12%

Lair Traces							
		Versioned Data	Versioned File Metadata	Versioned Directories	Version Ratio	Metadata Savings	Total Savings
Comprehensive:	Conventional	74.5 GB	34.8 GB	1.79 GB			
	CVFS	74.5 GB	1.1 GB	0.0064 GB	1:1	97%	32%
On close():	Conventional	40.3 GB	6.1 GB	0.75 GB			
	CVFS	40.3 GB	0.57 GB	0.0032 GB	1:2.9	91%	13%
6 minute Snapshots:	Conventional	38.2 GB	3.0 GB	0.75 GB			
	CVFS	38.2 GB	0.36 GB	0.0032 GB	1:11.2	88%	8%
1 hour Snapshots:	Conventional	36.2 GB	2.0 GB	0.75 GB			
	CVFS	36.2 GB	0.26 GB	0.0032 GB	1:15.6	87%	6%

Table 3: **Benefits for different versioning schemes.** This table shows the benefits of journal-based metadata for three versioning schemes that use pruning heuristics. For each scheme it compares conventional versioning with CVFS's journal-based metadata and multiversion b-trees, showing the versioned metadata sizes, the corresponding metadata savings, and the total space savings. It also displays the ratio of versions to file modifications; more modifications per version generally reduces both the importance and the compressibility of versioned metadata.

conventional versioning and journal-based metadata, the versioned data consumes the same amount of space, since both schemes use block sharing for versioned data. The overhead of versioned metadata is the information needed to track the versioned data. For *Labyrinth*, the versioned metadata consumes as much space as the versioned data. For *Lair*, it consumes only half as much space as the versioned data, because *Lair* uses a larger block size; on average, twice as much data is overwritten with each WRITE.

Journal-based metadata: Journal-based metadata reduces the space required for versioned file metadata substantially. For the conventional system, versioned metadata consists of copied inodes and sometimes indirect blocks. For journal-based metadata, it is the log entries that allow recreation of old versions plus any checkpoints used to improve back-in-time performance (see Section 5.3.2). For both traces, this results in 97% reduction of space required for versioned metadata.

Multiversion b-trees: Using multiversion b-trees for directories provides even larger space utilization reductions. Because directories are a metadata construct, there is no versioned data. The overhead of versioned metadata in directories is the space used to store the overwritten and deleted directory entries. In a conventional versioning system, each entry creation, modification, or removal results in a new block being written that contains the change. Since the entire block must be kept over the detection window, it results in approximately 9.7 GB of space for versioned entries in the *Labyrinth* trace and 1.8 GB in the *Lair* trace. With multiversion b-trees, the only overhead is keeping the extra entries in the tree, which results in approximately 45 MB and 7 MB of space for versioned entries in the respective traces.

5.2.1 Other Versioning Schemes

We also use the *Labyrinth* and *Lair* traces to compute the space that would be required to track versions in three other versioning schemes: versioning on every file CLOSE, taking systems snapshots every 6 minutes, and taking system snapshots every hour. In order to simulate open-close semantics with our NFS server, we insert a CLOSE call after sequences of operations on a given file that are followed by 500ms of inactivity to that file.

Table 3 shows the benefits of CVFS's mechanisms for the three versioning schemes mentioned above. For each scheme, the table also shows the ratio of file versions-to-modifications (e.g., in comprehensive versioning, each modification results in a new version, so the ratio is 1:1). For on-close versioning in the *Labyrinth* trace, conventional versioning requires 55% as much space for versioned metadata as versioned data, meaning that reduction can still provide large benefits. As the versioned metadata to versioned data ratio decreases and as the version ratio increases, the overall benefits of versioned metadata compression drop.

Table 3 identifies the benefits of both journal-based metadata (for "Versioned File Metadata") and multiversion b-trees (for "Versioned Directories"). For both, the metadata compression ratios are similar to those for comprehensive versioning. The journal-based metadata ratio drops slightly as the version ratio increases, because capturing more changes to the file metadata moves the journal entry size closer to the actual metadata size. The multiversion b-tree ratio is lower because a most of the directory updates fall into one of two categories: entries that are permanently added or temporary entries that are created and then rapidly renamed or deleted. For this reason, the number of versioned entries is lower for other versioning schemes; although multiversion b-trees use less space, the effect on overall savings is reduced.

5.3 Performance Overheads

The performance evaluation is done in three parts. First, we compare the S4 prototype to non-versioning systems using several macro benchmarks. Second, we measure the back-in-time performance characteristics of journal-based metadata. Third, we measure the general performance characteristics of multiversion b-trees.

5.3.1 General Comparison

The purpose of the general comparison is to verify that the S4 prototype performs comparably to non-versioning systems. Since part of our objective is to avoid undue performance overheads for versioning, it is important that we confirm that the prototype performs reasonably relative to similar systems. To evaluate the performance relationship between S4 and non-versioning systems, we ran two macro benchmarks designed to simulate realistic workloads.

For both, we compare S4 in both synchronous and asynchronous modes against three other systems: a NetBSD NFS server running FFS, a NetBSD NFS server running LFS, and a Linux NFS server running EXT2. We chose to compare against BSD's LFS because it uses a log-structured layout similar to S4's. BSD's FFS and Linux's EXT2 use similar, more "traditional" file layout techniques that differ from S4's log-structured layout. It is not our intent to compare a LFS layout against other layouts, but rather to confirm that our implementation does not have any significant performance anomalies. To ensure this, a small discussion of the performance differences between the systems is given for each benchmark.

Each of these systems was measured using an NFS client running on Linux. Our S4 measurements use the S4 server and a Linux client. For "Linux," we run RedHat 6.1 with a 2.2.17 kernel. For "NetBSD," we run a stock NetBSD 1.5 installation.

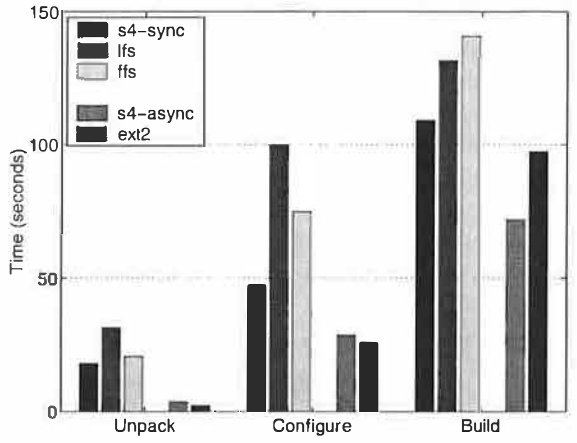


Figure 5: **SSH comparison.** This figure shows the performance of five systems on the unpack, configure, and build phases of the SSH-build benchmark. Performance is measured in the elapsed time of the phase. Each result is the average of 15 runs, and all variances are under .5s with the exception of the build phases of ffs and lfs which had variances of 37.6s and 65.8s respectively.

To focus comparisons, the five setups should be viewed in two groups. BSD LFS, BSD FFS, and S4-sync all push updates to disk synchronously, as required by the NFSv2 specification. Linux EXT2 and S4-async do not; instead, updates are made in memory and propagated to disk in the background.

SSH-build [39] was constructed as a replacement for the Andrew file system benchmark [20]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

Figure 5 shows the SSH-build results for each of the five different systems. As we hoped, our S4 prototype performs similarly to the other systems measured.

LFS does significantly worse on unpack and configure because it has poor small write performance. This is due to the fact that NetBSD's LFS implementation uses a 1 MB segment size, and NetBSD's NFS server requires a full sync of this segment with each modification; S4 uses a 64kB segment size, and supports partial segments. Adding these features to NetBSD's LFS implementation would result in performance similar to S4[1]. FFS performs worse than S4 because FFS must update both a data block and inode with each file modification, which are in separate locations on the disk. EXT2 performs more closely to S4 in asynchronous mode because it fails to satisfy NFS's requirement of synchronous modifications. It does slightly better in the unpack and configure stages because it maintains no consistency guarantees, however it does worse in the build phase due to S4's segment-sized reads.

Postmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [22]. It creates a large number of small randomly-sized files (between 512 B and 9 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction types are equal.

Figure 6 shows the Postmark results for the five server configurations. These show similar results to the SSH-build benchmark. Again, S4 performs comparably. In particular, LFS continues to perform poorly due to its small write performance penalty caused by its interaction with NFS. FFS still pays its performance penalty due to multiple updates per file create or delete. EXT2 performs even better in this benchmark because the random, small file accesses done in Postmark are not assisted by aggressive prefetching, unlike the sequential, larger accesses done during a compilation; however, S4 continues to pay the cost of doing larger accesses, while EXT2 does not.

5.3.2 Journal-based Metadata

Because the metadata structure of a file's current version is the same in both journal-based metadata and conventional versioning systems, their current version access times are identical. Given this, our performance measurements focus on the performance of back-in-time operations with journal-based metadata.

There are two main factors that affect the performance of back-in-time operations: checkpointing and clustering. Checkpointing refers to the frequency of metadata checkpoints. Since journal roll-back can begin with any checkpoint, CVFS keeps a list of metadata checkpoints for each file, allowing it to start roll-back from the closest checkpoint. The more frequently CVFS creates checkpoints, the better the back-in-time performance.

Clustering refers to the physical distance between relevant journal entries. With CVFS's log-structured layout, if several changes are made to a file in a short span of time, then the journal entries for these changes are likely to be clustered together in a single segment. If several journal entries are clustered in a single segment together, then they are all read together, speeding up journal roll-back. The "higher" the clustering, the better the performance is expected to be.

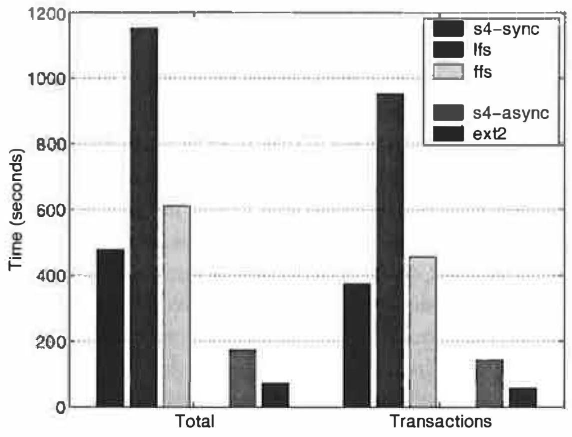


Figure 6: **Postmark comparison.** This figure shows the the elapsed time for both the entire run of postmark and the transactions phase of postmark for the five test systems. Each result is the average of 15 runs, and all variances are under 1.5s

Figure 7 shows the back-in-time performance characteristics of journal-based metadata. This graph shows the access time in milliseconds for a particular version number of a file back-in-time. For example, in the worst-case, accessing the 60th version back-in-time would take 350ms. The graph examines four different scenarios: best-case behavior, worst-case behavior, and two potential cases (one involving low clustering and one involving high clustering).

The best-case back-in-time performance is the situation where a checkpoint is kept for each version of the file, and so any version can be immediately accessed with no journal roll-back. This is the performance of a conventional versioning system. The worst-case performance is the situation where no checkpoints are kept, and every version must be created through journal roll-back. In addition there is no clustering, since each journal entry is in a separate segment on the disk. This results in a separate disk access to read each entry. In the high clustering case, changes are made in bursts, causing journal entries to be clustered together into segments. This reduces the slope of the back-in-time performance curve. In the low clustering case, journal entries are spread more evenly across the segments, giving a higher slope. In both the low and high clustering cases, the points where the performance drops back to the best-case are the locations of

[1]We tried changing the NetBSD LFS segment size, but it was not stable enough to complete any benchmark runs.

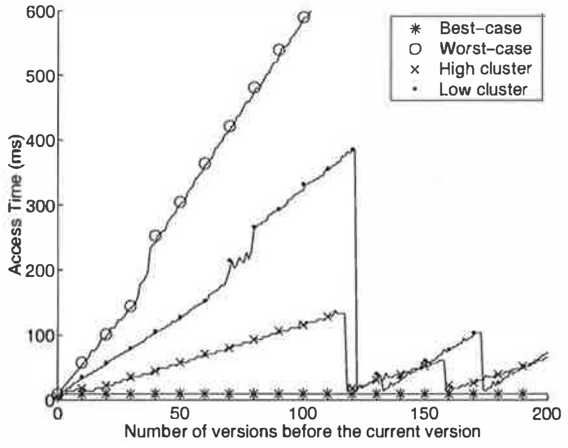


Figure 7: **Journal-based metadata back-in-time performance.** This figure shows several potential curves for back-in-time performance of accessing a single 1KB file. The worst-case is when journal roll-back is used exclusively, and each journal entry is in a separate segment on the disk. The best-case is if a checkpoint is available for each version, as in a conventional versioning system. The high and low clustering cases are examples of how checkpointing and access patterns can affect back-in-time performance. Both of these cases use random checkpointing. In the "high cluster" case, there are an average of 5 versions in a segment. In the "low cluster" case, there are an average of 2 versions in a segment. The cliffs in these curves indicate the locations of checkpoints, since the access time for a checkpointed version drops to the best-case performance. As the level of clustering increases, the slope of the curve decreases, since multiple journal entries are read together in a single segment. Each curve is the average of 5 runs, and all variances are under 1ms.

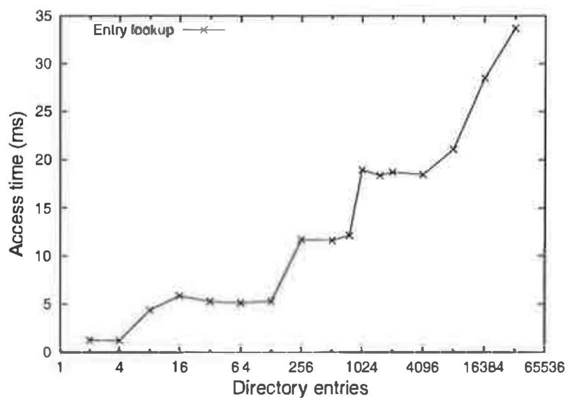


Figure 8: **Directory entry performance.** This figure shows the average time to access a single entry out of the directory given the total number of entries within the directory. History entries affect performance by increasing the effective number of entries within the directory. The larger the ratio of history entries to current entries, the more current version performance will suffer. This curve is the average of 15 runs and the variance for each point is under .2ms.

checkpoints.

Using this knowledge of back-in-time performance, a system can perform a few optimizations. By tracking checkpoint frequency and journal entry clustering, CVFS can predict the back-in-time performance of a file while it is being written. With this information, CVFS bounds the performance of the back-in-time operations for a particular file by forcing a checkpoint whenever back-in-time performance is expected to be poor. For example, in Figure 7, the high-clustering case keeps checkpoints in such a way as to bound back-in-time performance to around 100ms at worst. In our S4 prototype, we bound the back-in-time performance to approximately 150ms. Another possibility is to keep checkpoints at the point at which one believes the user would wish to access the file. Using a heuristic such as in the Elephant FS [37] to decide when to create file checkpoints might closely simulate the back-in-time performance of conventional versioning.

5.3.3 Multiversion B-trees

Figure 8 shows the average access time of a single entry from a directory given some fixed number of entries currently stored within the directory (notice the log scale of the x-axis). To see how a multiversion b-tree performs as compared to a standard b-tree, we must compare two different points on the graph. The point on the graph corresponding to the number of current entries in the directory represents the access time of a standard b-tree. The point on the graph corresponding to the combined number of current and history entries represents the access time of a multiversion b-tree. The difference between these values is the lookup performance lost by keeping the extra versions.

Using the traces gathered from our NFS server, we found that the average number of current entries in a directory is approximately 16. Given a detection window of one month, the number of history entries is less than 100 over 99% of the time, and between zero and five over 95% of the time. Since approximately 200 entries can fit into a block, there is generally no performance lost by keeping the history. This block-based performance explains the stepped nature of Figure 8.

5.4 Summary

Our results show that CVFS reduces the space utilization of versioned metadata by more than 80% without causing noticeable performance degradation to current version access. In addition, through intelligent checkpointing, it is possible to bound back-in-time performance to within a constant factor of conventional versioning systems.

6 Metadata Versioning in Non-Log-Structured Systems

Most file systems today use a layout scheme similar to that of BSD FFS rather than a log-structured layout. Such systems can be extended to support versioning relatively easily; Santry et al. [37] describe how this was done for Elephant, including tracking of versions and block sharing among versions. For non-trivial pruning policies, such as those used in Elephant and CVFS, a cleaner is required to expire old file versions. In an FFS-like system, unlike in LFS, the cleaner does not necessarily have the additional task of coalescing free space.

Both journal-based metadata and multiversion b-trees can be used in a versioning FFS-like file system in much the same way as in CVFS. Replacing conventional directories with multiversion b-trees is a self-contained change, and the characteristics should be as described in the paper. Replacing replicated metadata versions with journal-based metadata requires effort similar to adding write-ahead logging support. Experience with adding such support [17, 39] suggests that relatively little effort is involved, little change to the on-disk structures is involved, and substantial benefits accrue. If such logging is already present, journal-based metadata can piggyback on it.

Given write-ahead logging support, journal-based metadata requires three things. First, updates to the write-ahead log, which are the journal entries, must contain enough information to roll-back as well as roll-forward. Second, the journal entries must be kept until the cleaner removes them; they cannot be removed via standard log checkpointing. This will increase the amount of space required for the log, but by much less than the space required to instead retain metadata replicas. Third, the metadata replica support must be retained for use in tracking metadata version checkpoints.

With a clean slate, we think an LFS-based design is superior if many versions are to be retained. Each version committed to disk requires multiple updates, and LFS coalesces those updates into a single disk write. LFS does come with cleaning and fragmentation issues, but researchers have developed sufficiently reasonable solutions to them to make the benefits outweigh the costs in many environments. FFS-type systems that employ copy-on-write versioning have similar fragmentation issues.

7 Related Work

Much work has been done in the areas of versioning and versioned data structures, log-structured file systems, and journaling.

Several file systems have used versioning to provide recovery from both user errors and system failure. Both Cedar [17] and VMS [29] use file systems that offer simple versioning heuristics to help users recover from their mistakes. The more recent Elephant file system provides a more complete range of versioning options for recovery from user error [37]. Its heuristics attempt to keep only those versions of a file that are most important to users.

Many modern systems support snapshots to assist recovery from system failure [11, 19, 20, 25, 34]. Most closely related to CVFS are Spiralog [15, 21] and Plan9 [33], which use a log-structured file system to do online backup by recording the entire log to tertiary storage. Chervenak, et al., performed an evaluation of several snapshot systems [10].

Version control systems are user programs that implement a versioning system on top of a traditional file system [16, 27, 43]. These systems store the current version of the file, along with differences that can be applied to retrieve old versions. These systems usually have no concept of checkpointing, and so recreating old versions is expensive.

Write-once storage media keeps a copy of any data written to it. The Plan 9 system [33] utilized this media to permanently retain all filesystem snapshots using a log-structured technique. A recent improvement to this method is the Venti archival storage system. Venti creates a hash of each block written and uses that as a unique identifier to map identical data blocks onto the same physical location [34]. This removes the need to rewrite identical blocks, reducing the space required by individual data versions and files that contain similar data. It is interesting to consider combining Venti's data versioning with CVFS's metadata structures to provide extremely space efficient comprehensive versioning.

In addition to the significant file system work in versioning, there has been quite a bit of work done in the database community for keeping versions of data through time. Most of this work has been done in the form of "temporal" data structures [2, 23, 24, 44, 45]. Our directory structure borrows from these techniques.

The log-structured data layout was developed for write-once media [33], and later extended to provide write performance benefits for read-write disk technology [36]. Since its inception, LFS has been evaluated [3, 28, 35, 38] and used [1, 7, 12, 18] by many different groups. Much of the work done to improve both LFS and LFS cleaners is directly applicable to CVFS.

While journal-based metadata is a new concept, journaling has been used in several different file systems to provide metadata consistency guarantees efficiently [8, 9, 11, 39, 42]. Similarly to journal-based metadata, LFS's segment summary block contains all of the metadata for the data in a segment, but is stored in an uncompressed format. Zebra's deltas improved upon this by storing only the changes to the metadata, but were designed exclusively for roll-forward (a write-ahead log). Database systems also use the roll-back and roll-forward concepts to ensure consistency during transactions with commit and abort [14].

Several systems have used copy-on-write and differencing techniques that are common to versioning systems to decrease the bandwidth required during system backup or distributed version updates [4, 6, 26, 31, 32]. Some of these data differencing techniques [5, 26, 31] could be applied to CVFS to reduce the space utilization of versioned data.

8 Conclusion

This paper shows that journal-based metadata and multiversion b-trees address the space-inefficiency of conventional versioning. Integrating them into the CVFS file system has nearly doubled the detection window that can be provided with a given storage capacity. Further, current version performance is affected minimally, and back-in-time performance can be bounded reasonably with checkpointing.

Acknowledgments

We would like to thank our shepard Mike Franklin and the anonymous reviewers for their comments and helpful suggestions. Also, Dan Ellard and others at Harvard for providing the *Lair* trace used in this work. We thank the members and companies of the PDL Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We thank IBM and Intel for hardware grants supporting our research efforts. Garth Goodson has been supported by an IBM Fellowship. Craig Soules has been supported by a Usenix Association Scholarship. This material is based in part on research sponsored by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, and by the DARPA/ITO OASIS program (Air Force contract number F30602-99-2-0539-AFRL.[2]

[2]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5):109–126, 1995.

[2] B. Becker, S. Gschwind, T. Ohler, P. Widmayer, and B. Seeger. An asymptotically optimal multiversion b-tree. *Very large data bases journal*, **5**(4):264–275, 1996.

[3] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. USENIX Annual Technical Conference, pages 277–288. USENIX Association, 1995.

[4] R. C. Burns. Version management and recoverability for large object data. International Workshop on Multimedia Database Management, pages 12–19. IEEE Computer Society, 1998.

[5] R. C. Burns. *Differential compression: a generalized solution for binary files*. Masters thesis. University of California at Santa Cruz, December 1996.

[6] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. Workshop on Input/Output in Parallel and Distributed Systems, pages 26–36. ACM Press, December 1997.

[7] M. Burrows, C. Jerian, B. Lampson, and T. Mann. *Online data compression in a log-structured file system*. 85. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, April 1992.

[8] L. F. Cabrera, B. Andrew, K. Peltonen, and N. Kusters. Advances in Windows NT storage management. *Computer*, **31**(10):48–54, October 1998.

[9] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, **34**(1):105–110, January 1990.

[10] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: a survey of backup techniques. Joint NASA and IEEE Mass Storage Conference, 1998.

[11] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. USENIX Annual Technical Conference, pages 43–60, 1992.

[12] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: a new approach to improving file systems. ACM Symposium on Operating System Principles, pages 15–28, 1993.

[13] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of an Email and Research Workload. Conference on File and Storage Technologies. USENIX Association, 2003.

[14] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Potzulo, and I. Traiger. The recovery manager of the System R database manager. *ACM Computing*

Surveys, **13**(2):223–242, June 1981.

[15] R. J. Green, A. C. Baird, and J. Christopher. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, **8**(2):32–45, 1996.

[16] D. Grune, B. Berliner, and J. Polk. Concurrent Versioning System, http://www.cvshome.org/.

[17] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **21**(5):155–162, 1987.

[18] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, **13**(3):274–310. ACM Press, August 1995.

[19] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Winter USENIX Technical Conference, pages 235–246. USENIX Association, 1994.

[20] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[21] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, **8**(2):5–14, 1996.

[22] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[23] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering*, **10**(1), February 1998.

[24] S. Lanka and E. Mays. Fully Persistent B+-trees. ACM SIGMOD International Conference on Management of Data, pages 426–435. ACM, 1991.

[25] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[26] J. MacDonald. *File system support for delta compression*. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[27] J. MacDonald, P. N. Hilfinger, and L. Semenzato. PRCS: The project revision control system. European Conference on Object-Oriented Programming. Published as *Proceedings of ECOOP*, pages 33–45. Springer-Verlag, 1998.

[28] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **31**(5):238–252. ACM, 1997.

[29] K. McCoy. *VMS file system internals*. Digital Press, 1990.

[30] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–197, August 1984.

[31] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. ACM Symposium on Operating System Principles. Published as *Operating System Review*, **35**(5):174–187. ACM, 2001.

[32] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. Conference on File and Storage Technologies, pages 117–129. USENIX Association, 2002.

[33] S. Quinlan. A cached WORM file system. *Software—Practice and Experience*, **21**(12):1289–1299, December 1991.

[34] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies, pages 89–101. USENIX Association, 2002.

[35] J. T. Robinson. Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning. *Operating Systems Review*, **30**(4):29–32, October 1996.

[36] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **25**(5):1–15, 1991.

[37] D. S. Santry, M. J. Feeley, N. C. Hutchinson, R. W. Carton, J. Ofir, and A. C. Veitch. Deciding when to forget in the Elephant file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **33**(5):110–123. ACM, 1999.

[38] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: a performance comparison. USENIX Annual Technical Conference, pages 249–264. Usenix Association, 1995.

[39] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. USENIX Annual Technical Conference, 2000.

[40] J. D. Strunk, G. R. Goodson, A. G. Pennington, C. A. N. Soules, and G. R. Ganger. *Intrusion detection, diagnosis, and recovery with self-securing storage*. Technical report CMU–CS–02–140. Carnegie Mellon University, 2002.

[41] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation, pages 165–180. USENIX Association, 2000.

[42] A. Sweeney. Scalability in the XFS file system. USENIX Annual Technical Conference, pages 1–14, 1996.

[43] W. F. Tichy. *Software development control based on system structure description*. PhD thesis. Carnegie-Mellon University, Pittsburgh, PA, January 1980.

[44] V. J. Tsotras and N. Kangelaris. The snapshot index - an I/O-optimal access method for timeslice queries. *Information Systems*, **20**(3):237–260, May 1995.

[45] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, **9**(3). IEEE, May 1997.

yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking*

Zhihui Zhang and Kanad Ghose
Department of Computer Science,
State University of New York, Binghamton, NY 13902-6000
e-mail: {zzhang, ghose}@cs.binghamton.edu

Abstract

In recent years, disk seek times have not improved commensurately with CPU performance, memory system performance, and disk transfer rates. Furthermore, many modern applications are making increasing use of large files. Traditional file system designs are limited in how they address these two trends. We present the design of a file system called yFS that consciously reduces disk seeking and handles large files efficiently. yFS does this by using extent-based allocations in conjunction with three different disk inode formats geared towards small, medium, and large files. Directory traversals are made efficient by using the B*-tree structure. yFS also uses lightweight asynchronous journaling to handle metadata changes. We have implemented yFS on FreeBSD and evaluated it using a variety of benchmarks. Our experimental evaluations show that yFS performs considerably better than the Fast File System (FFS) with Soft Updates on FreeBSD. The performance gains are in the range from 20% to 82%.

1 Introduction

Recent years have seen impressive growth in CPU speeds and hard drive technology. In particular, data transfer rates of hard drives have improved quite dramatically with an increase in the performance of the data transfer paths and the rotational speeds [1]. However, processing capabilities have improved even more than disk access rates. At the same time, the demands for handling intensive I/O and large data sets have also grown noticeably [2]. These trends indicate that the I/O bottleneck will continue to be an issue in the foreseeable future.

This paper describes our effort in creating a new file system, named yFS, to alleviate this I/O bottleneck. yFS is a new file system design that brings together many of the best-known techniques for improving file system performance. By starting from scratch, we have been able to integrate proven techniques and new ideas with greater freedom and flexibility compared to either porting or patching an existing file system.

*This work is supported in part by the NSF through award No. EIA 0011099.

We have chosen FreeBSD as the host operating system of yFS because FreeBSD is a mature and well-pedigreed open source operating system [3]. Specifically, it has a solid merged buffer cache design and a full-fledged VFS/vnode architecture. Both features have directly influenced the design of yFS.

The rest of this paper is organized as follows. Section 2 discusses our motivation for this research. Section 3 describes the organization of yFS. Section 4 outlines the journaling model of yFS. Section 5 describes the implementation of yFS. Section 6 presents our experimental evaluation results. Section 7 discusses related work. We conclude this paper in Section 8.

2 Motivation

File system designs have always been driven by changes in two major arenas: hardware and workloads. Traditional file systems such as the Fast File System (FFS) [4] and similar file systems (e.g., Ext2 file system [5]) were designed with different assumptions about the underlying hardware and the workloads to which the file systems would be subjected. Although still used widely, there are a number of known techniques (e.g., journaling, B*-tree indices, extent-based allocation) that are not used in these systems and have been shown to perform better [5,6,7,8]. Some limitations in these file system designs are discussed below.

In classical file system designs, one block is usually allocated at a time even for a multi-block I/O request. Because traditional designs do not maintain sufficient information about the availability of extents, the preferred block is the one contiguous to a previously allocated block, even if there may be a larger extent available elsewhere. FFS tries to alleviate this problem by using a separate cluster map and a reallocation step [9], but it introduces the complexity of maintaining two bitmaps and each reallocation step must involve all the buffers in the cluster.

File systems that provide only a single allocation unit size force one to make a trade-off between performance and fragmentation. FFS divides a single file system block into one or more *fragments*, so that it can avoid undue internal fragmentation on small files and boost I/O throughput for large ones. However, FFS requires that a block consist of at most eight fragments and start at fixed fragment addresses. This can lead to poor file data layout and wasted disk space. As a pathological example, if most of the files are a little more than 4096 bytes in an 8192/1024 file system, then almost half of the disk space will be wasted.

Traditional file systems translate a logical block number to its corresponding physical block numbers using file-specific metadata in the form of a highly skewed tree [4]. For large files, several indirections are needed before a data block can be accessed; this is because only a few direct pointers are stored inside the disk inode. Furthermore, although one could represent a series of contiguously allocated blocks efficiently with a single disk address, conventional block-based systems do not do so. The VIVA file system [10,11] reduces these indirections by compressing disk block addresses with partial bitmaps, but fails to address the problem of holes efficiently.

Metadata integrity is crucial in a file system. Historically, metadata was updated synchronously in a pre-determined order to ease the job of *fsck* [12]. This not only severely limits the performance of metadata updates, but also entails scanning of the entire file system for crash recovery. Log-structured File Systems (LFSs) solve both problems with the technique of logging [13,14]. LFS treats the disk as a segmented append-only log and writes both the file data and the metadata into it. Crash recovery can be performed efficiently by taking the most recent checkpoint and proceeding to the tail of the log. However, LFS introduces cleaning overhead, since the size of the file system is of finite size and the log must eventually wrap. Metadata journaling is a second technique that logs only changes to the metadata [6,7,8]. It speeds up metadata updates and crash recovery without incurring any of the cleaning cost of LFSs. However, metadata journaling introduces extra logging I/O because the metadata has to be written twice—first in the log area and then in place. Soft Updates is a third technique used to tackle the metadata update problem [15,16,17]. It maintains metadata dependency information at per-pointer granularity in the memory so that delayed writes can be used safely for metadata updates. To avoid dependency cycles, any still-dependent updates in a metadata block are rolled-back before I/O and rolled-forward after I/O. After a system crash, *fsck* is still needed to salvage any unused blocks and inodes.

Traditional file systems maintain a directory as an unsorted linear list of file name to inode number mappings called *directory entries*, which is painful to handle large directories. In addition, some old file systems lack the ability to create and deallocate new disk inodes on the fly.

To summarize, traditional file systems have various weaknesses in some or all of these subjects: disk space allocation, large directory handling, dynamic inode allocation, metadata update performance, and fast crash recovery. The goal of our work on yFS is to create a new file system that handles these issues more efficiently. Specifically, yFS has the following features:

- yFS uses extent-based allocation to maximize the chances of contiguous allocation.
- yFS makes use of B*-trees for representing large files and directories.
- yFS uses a variety of techniques to reduce the overhead of metadata logging to achieve fast crash recovery and boost metadata update performance.
- yFS allows dynamic allocation and deallocation of disk inodes.
- yFS implements fragments and inline data storage to improve the performance on small files.

Modern file systems like IBM JFS [6], SGI XFS [7,8], and FFS with Soft Updates [15,16,17] technology have also made improvements in solving design problems found in traditional file systems. In Section 7, we will compare yFS with these more modern file system designs.

3 The Organization of yFS

In this section, we discuss the major data structures of yFS and the disk space allocation strategy of yFS.

3.1 Allocation Groups

yFS divides the file system disk space into equal-sized units called allocation groups (AGs). The purpose of an AG is similar to that of cylinder groups in the Fast File System [4], that is, to cluster related data. In addition, we use AGs to increase concurrency by allowing different processes to work in different AGs. An AG consists of a superblock copy, AG group block, AG inode block, AG bitmap block, a set of preallocated inodes, and a large number of available data blocks.

yFS makes use of a bitmap to keep track of disk space usage within an AG. Like FFS, we use the idea of "sub-blocking" to trade off between space efficiency and I/O throughput. Each bit in the AG bitmap block represents one fragment, which is the smallest allocation unit. In addition, yFS also has a block size. A block is composed of one or more fragments and *can start at any fragment address*. In yFS, large files (with a size of more than 12 blocks) are always allocated in full blocks, while the last block of small files are allocated as many fragments as needed. As in FFS, fragment reallocation is needed if the last block of a small file cannot grow in place.

The AG group block and the AG inode block contain disk space and disk inode information respectively. To allocate disk space or a disk inode from a particular AG, we must first lock the buffer of its group block or its inode block. As explained in Section 4, every atomic operation performed in yFS is implemented as a transaction. Because a transaction does not release locks on metadata buffers before it commits, it should not allocate disk blocks or disk inodes from two AGs at the same time to avoid any potential deadlock. The separation of disk space and disk inode information into two different AG blocks (AG group block and AG inode block) is deliberate. It allows allocation of disk space in one AG and a disk inode in another AG within the same transaction.

3.2 Extent Summary Tree

To avoid a linear search on a bare-bones bitmap, we have adapted the IBM JFS algorithm which uses a binary buddy encoding of the bitmap and a free-extent summary tree built on top of the encoding [6]. The binary buddy representation of the bitmap and the summary tree are stored in the AG group block, separate from the AG bitmap block.

Initially, we divide the entire bitmap into 32-bit chunks called *bitmap groups*. Each group is then encoded using the binary buddy scheme (e.g., a value of 4 means that the maximum buddy group size within this bitmap group is 4 and it contains a free extent of length 2^4). The encoded value of a bitmap group can be stored in *one byte* because its maximum value is 5. After encoding bitmap groups, we merge buddies among groups if possible. During each merge, the left buddy's encoded value is incremented, while its right buddy's encoded value is *negated*. For example, if two buddies with an encoded value of 5 are merged, the left buddy will have an encoded value of 6 and the right buddy will have an encoded value of –5. If a bitmap group does not have any free fragments at all, it is assigned a special value of NOFREE (–128).

The binary buddy representation of the bitmap can be used to check the availability of free fragments without examining the underlying bitmap itself. For example, if we know the encoding value of a buddy group is 13, then we immediately know that all 2^{13} fragments starting from the first fragment of this group are free.

The extent summary tree is used to find out if and where a free extent of size 2^n is available. It is simply a 4-ary tree with each parent taking the maximum value of its four children. Each leaf in the tree takes on the binary buddy encoding of its corresponding bitmap group. This tree can be constructed bottom-up easily. Because the size of the bitmap is fixed (and hence the number of bitmap groups), the size of the tree is also fixed and can be represented using a flat array. Each node in the tree occupies one byte, capable of representing a maximum free extent of 2^{127} fragments. When we search the extent summary tree, we use the *absolute* values of its nodes to determine whether there is free space covered by a node unless, of course, the value is NOFREE.

The free-extent summary tree can locate the portion of the bitmap with enough free space quickly. However, the binary buddy encoding only records the availability of free extents of size 2^n, which must begin at a fragment address that is a multiple of 2^n. This could lead to sub-optimal allocations if we used the summary tree liberally. As a result, we consult the summary tree only when it is advantageous to do so (Section 3.3). An important note here is that we can always work on the bitmap directly and then adjust the encoding information accordingly. It is also important to know that we do not have to allocate an extent at the beginning of a buddy group. If the bitmap group from where we want to start an allocation has a negative encoding value, we must first reverse the effects of previous buddy merges (i.e., perform back split) until the bitmap group gets its free fragments back from its left buddy (or buddies).

3.3 Disk Resource Allocation

Disk resource (including disk inodes and disk blocks) allocation is the single most important issue in any file system design. yFS uses a two-step approach to perform this task. First, it chooses an AG. Second, it allocates resources from the chosen AG. To improve performance, yFS pursues two goals during disk resource allocation: *locality* and *contiguity*.

Many file systems use concepts similar to AGs to localize related data and spread data across the file system. For example, the Solaris file system achieves good temporal locality by forcing all allocations into one hot-spot allocation group [18]. The original FFS algorithm places a directory into a different cylinder group of its parent directory to ensure logical locality [4]. Both of these strategies have their weaknesses. Trying to cluster too much file data in the same AG exhausts its space quickly. This over-localization prevents future related files from being stored locally. On the other hand, switching to a new AG for each directory incurs disk seeks whenever we need to move along the directory hierarchy.

yFS adopts an algorithm similar to that used in FFS of FreeBSD 4.5 to choose an AG for non-directory files and directory files [19]. A non-directory file is preferred to be created in the same AG as its parent directory. A file system-wide parameter *maxblkpg* (maximum blocks per AG) is used to spread data blocks of a large file across AGs. Directories are laid out in a different way using the idea of "directory space reservation." Basically, a system administrator can provide two parameters when creating a file system: the average file size and the average number of files per directory. The disk space needed for each directory is reserved whenever possible (i.e., the free space in an AG should not drop well below the average amount of free space among all the AGs) using the above two parameters to avoid spreading out directories too aggressively. As a result, more directories can be created in the same AG with enough space for files to be created within them. We thus are able to take care of both temporal and logical locality in disk space allocation.

The behavior of the resource allocator can be modified by its various internal callers using bit flags. For example, the flag ALLOC_ANYWHERE is set when we want to allocate disk blocks for a file. This flag allows the allocator to search the entire file system for available disk blocks. However, if a caller wants to create more disk inodes in a particular AG, it will use the flag ALLOC_GROUP to limit the allocation within that AG.

After a usable AG is chosen, the preferred location within that AG (i.e., hint within the AG) can be determined if previous allocation information is available to indicate where contiguous allocation is possible. Allocation then proceeds as follows:

(1) If we have a hint within the AG, we attempt to use it before looking up the free-extent summary tree. This guarantees that files are allocated contiguously whenever possible. If there is a hole between two successive writes, the hint is adjusted to reserve space for the hole. This retains the possibility that when the hole is filled later, the entire file remains contiguous.

(2) Otherwise, we try the location of the first free fragment in the AG. If this fails, we then try the location following that for the last successful allocation, failing which we search the free-extent summary tree as the last resort.

yFS uses extent-based allocation to allocate more than one fragment per call whenever possible. The size of a newly allocated extent depends on the current I/O size as well as the availability of free space. It is thus possible for a large allocation request to be satisfied by several disjoint extents, possibly from different AGs. This is different from file systems that use a pre-determined extent size—VxFS is one example of such a file system [20]. As the file system fills up, the availability of large free extents decreases. Even so, yFS can locate free extents quickly with the aid of the free-extent summary tree. Note that the size of an extent is counted in fragments, not in blocks.

An extent allocated to a file is naturally described by an *extent descriptor* that is shown in Figure 1(a). Note that we may need more than one descriptor to describe one physical extent if the extent is not contiguous in terms of its logical block numbers. On the other hand, two extents can be merged if the hole between them is filled later by data from the same file.

3.4 Ubiquitous B*-tree

yFS uses B*-trees in three different contexts for: (a) disk inode management for each AG; (b) extent descriptor management for files with a large number of extents; (c) directory block descriptor management for directories with a large number of directory blocks.

The power of B*-trees lies in the fact that they are shallow and well-balanced. B*-trees give the capability of performing localized structural changes and guarantee at least 50% storage utilization [21]. These B*-tree features are indispensable in supporting any large and dynamic data set on a secondary storage device. The algorithms associated with B*-trees are complicated, so we use them only when necessary. The size of B*-tree nodes in yFS is the same as the full block size. However, the root node embedded in a disk inode is much smaller.

Each disk inode in yFS can be in one of the following three formats, as shown in Figures 1(b) through 1(d):

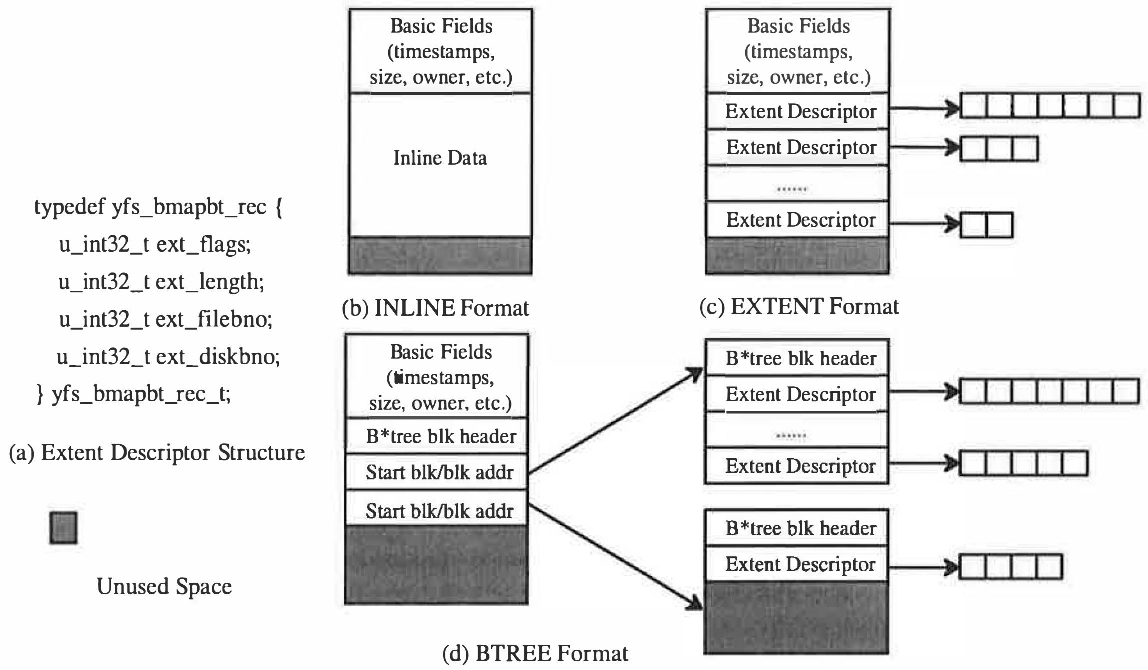
INLINE. For a small enough file, all its data can be stored inside its disk inode. This format reduces the number of seeks needed to access small files, because all metadata and data are stored in one place, namely, within the inode.

EXTENT. Here the extent descriptors of a regular file or directory block descriptors (Section 3.5) of a directory file are stored as an array that can fit into the disk inode. Note that even a large file can use this format if it has only a few non-contiguous extents.

BTREE. For a file with more descriptors than can fit into its disk inode, its descriptors are organized into a B*-tree. We always store the root block of its B*-tree in the inode, reducing the worse case search by one disk access as long as the file remains open.

The formats for directory inodes are similar except that we use directory block descriptors instead of extent descriptors. In addition to B*-trees used by files, each AG has its own disk inode B*-tree that is used to support allocation and deallocation of disk inodes on the fly within it. The root node of the disk inode B*-tree is always stored in the AG inode block. Since it is not embedded in a disk inode, its size is the block size.

In yFS, inodes are 512 bytes (one sector), which are four times larger than those of FFS. We have done this for three reasons: (1) It allows us to use sector addresses as inode numbers directly; as a result, we do not have to look up the disk inode B*-tree to find the disk address of a given inode. (2) We have better support for small files with inline format. In our prototype, at most 420 bytes of data can be stored inline; as a result, many small script files and small directories can be stored efficiently and accessed cheaply. (3) Since an inode block contains more than one disk inode, using a larger inode size reduces lock competition of the same disk inode block.



```
typedef yfs_bmapbt_rec {
    u_int32_t ext_flags;
    u_int32_t ext_length;
    u_int32_t ext_filebno;
    u_int32_t ext_diskbno;
} yfs_bmapbt_rec_t;
```

Figure 1. Regular File Disk Inode Formats

In yFS, although more I/O operations will be performed to bring in the same number of disk inodes, we feel that the above advantages outweigh this disadvantage.

3.5 Large Directory Support

The directory design of yFS aims to provide efficient support for both small and large directories. Unlike classical file system designs that implement a directory as an unsorted list of directory entries, all entries in yFS are ordered by their hash values computed from their corresponding file names using simple left rotation and XOR operations. If a directory is small, all its entries can be stored inside its disk inode (inline format). Otherwise, directory entries will be stored in *directory blocks*, which are described by *directory block descriptors*. The descriptor records three kinds of information for a directory block: address, smallest and the biggest hash values of all its entries, and space usage. All the directory block descriptors of a directory are stored in the disk inode if the number is small enough to fit into it (extent format) or organized into a B*-tree using smallest hash values as keys (Btree format). For the Btree format, all the B*-tree leaf blocks are linked together to support the *getdents()* system call easily.

yFS uses fixed 32-bit hash values as the keys for a directory B*-tree. Compared to using variable sized keys involved in techniques such as suffix compression [6,22], using fixed size keys greatly reduces complexity and enables binary search of keys within a B*-tree block. To cope with duplicate keys, any binary search result must be adjusted so that the first of a sequence of duplicate keys is always chosen at each level of a B*-tree. The rest of the directory code must also be able to deal with this situation.

Traditional file systems always use simple offsets as the directory cookies. In yFS, directory cookies are 64 bits, whose contents depend on the format of the directory inode. For a small inline directory, it is simply an offset to the next directory entry. For an extent format directory, the directory cookie consists of an index into the array of directory block descriptors stored in the disk inode and an offset into the directory block pointed to by the chosen directory descriptor. For a B*-tree format directory, the directory cookie consists of three components: the block address of a leaf B*-tree block, an index into the leaf block to select a directory block descriptor, and an offset into the directory block pointed to by the selected directory block descriptor.

The recently published design of Htree for Linux [23] also proposes the use of hash-keyed B*-trees. yFS differs from this effort in at least two aspects: (1) using a sorted linked list of entries to improve logging performance (Section 5), and (2) using physical addressing instead of logical addressing in the B*-tree to reduce one level of indirection.

4 Metadata Journaling

Like many other file systems [6,7,17], yFS uses metadata journaling to improve metadata update performance and provide fast recovery after a system failure. However, metadata journaling can rapidly become the system's performance bottleneck [17], so we take great care to make our logging system lightweight.

4.1 Transaction Considerations

A file system journaling design should be based on a good understanding of file system semantics and requires consideration of the following issues:

(1) Although a user activity (e.g., creating a file) is the ultimate source for triggering a transaction, it is up to the file system to determine when to start a transaction and what specific actions make up the transaction.

(2) A transaction in a file system is typically small. For example, a transaction used to change the owner of a file need only modify one inode block. For some large atomic operations, we can separate them into small transactions with the help of extra mechanisms (Section 4.4). As a result, all transactions run entirely within memory using the no-steal policy [22], which means that dirty buffers will not be reclaimed until the corresponding transaction commits in-core.

(3) Although we log only metadata changes, it is dangerous to sever the relationship between the file data and its associated metadata. As an example, if a metadata block is freed and then reused for file data prematurely, an untimely crash can corrupt the file system. Our solution to this problem (Section 4.4) neither restricts the reuse of freed metadata blocks [24] nor requires the use of any additional committed bitmap to check if a block is allocatable [5].

(4) We cannot afford to abort a dirty transaction for the sake of performance. In GFS [25], whenever a transaction modifies a buffer, a copy is made to preserve its old contents. If the transaction must be aborted, GFS simply restores all affected buffers by using their frozen copies. Such a scheme is expensive in terms of its memory footprint and copying overhead. As a result, we must make sure that once started, a transaction will succeed. We achieve this through the use of resource reservations.

(5) The whole purpose of using the transaction mechanism is to guarantee the integrity of the file system after a crash. In yFS, we do not attempt to provide a history of updates that allows us to roll back to some point in the past. This means that log space can be reclaimed as soon as the metadata it protects has been written in place.

The above considerations lead to a lightweight transaction model for yFS. The implementation of this model has a small impact on performance.

4.2 Transaction Model: Overview

yFS uses a variety of features to improve the efficiency of its metadata transactions. These features include: (1) fine-granularity logging, (2) dynamic incore log buffers, (3) asynchronous group commit, (4) resource reservations, (5) background daemons, and (6) a circular log area.

We first define a transaction as a group of metadata modifications that must be carried out atomically. A transaction moves the file system from one consistent state to another consistent state.

In yFS, all metadata changes are made by use of metadata buffers. Whenever a transaction wants to access a metadata buffer, a *log item* is created for the buffer if it does not already have one. A log item records if and where its associated metadata buffer is modified. Each transaction keeps track of the buffers it has locked indirectly by maintaining log item pointers.

When a transaction ends, all metadata changes made by it are first copied to incore log buffers in the form of log entries. yFS uses physical logging [22] because of its simplicity and idempotence. Each log entry is created from the information recorded in a log item and consists of: a metadata block number; start and end offsets within the block where the block was modified; new data for that region of the block; and a transaction ID. A special commit log entry is used to indicate if all log entries of a transaction have been written. After this, the transaction is said to have committed in core. Although we use the two-phase locking protocol to achieve transaction isolation, we never hold a buffer lock across an I/O operation, as in XFS [7,8]. After a transaction is committed in core, all its buffers will be unlocked immediately. However, a modified buffer is pinned in memory to enforce the Write-Ahead Logging (WAL) protocol [22]. A hot metadata buffer (e.g., a bitmap block) can be pinned by more than one transaction: a pin count is used in this case to indicate the number of non-committed transaction for this buffer. As a transaction commits on-disk, the associated buffer's pin count is decremented; a buffer is reclaimed only when its pin count is zero.

4.3 Transaction Model: Details

Figure 2 depicts the key data structures related to transaction handling in yFS. All log entries, usually from different transactions, stored in an incore log buffer constitute one *log record*. A log record is checksumed and timestamped. The size of a log record is unknown until the incore log buffer is full or flushed before it is full, and it is always rounded up to a multiple of the sector size. A 64-bit Log Sequence Number (LSN) is assigned to an incore log buffer (which contains one log record) when it is written for the first time. The LSN consists of two parts: a cycle number that is incremented each time the on-disk log wraps around and a sector address where the log record should be stored on disk. A transaction's LSN is the LSN of the incore log buffer that contains its commit log entry (in Figure 2, transaction #123 will have LSN of 5678).

All incore log buffers are organized into two queues: the idle queue and the active queue. Whenever a new incore log buffer is used, it is moved from the idle queue to the tail of the active queue. An incore log buffer is flushed by a special *log daemon*, triggered by either a log buffer becoming full, or a 30 second timer, or a synchronous operation. After an incore log buffer is flushed to the disk, transactions that have written a commit log entry into it are group committed onto the disk and the incore log buffer is moved back to the idle queue for reuse. If a flushed incore log buffer is not at the head of the active queue, these processing steps are deferred until its preceding incore log buffers are flushed and processed. This guarantees that transactions always commit on disk in the same order that they commit in core. To accommodate different rates of metadata updates, the number of incore log buffers can be adjusted dynamically.

A transaction's lifetime terminates when it is committed onto the disk. In other words, the changes made by the transaction are now made permanent. But before it dies, all the buffers dirtied by it are unpinned and their associated log items are tagged with the transaction's LSN indicating where the modified data is logged. However, if a log item already has a smaller LSN, its LSN should not be replaced with a bigger one because it must remember the first log sector that has active log data for its associated metadata buffer. Note that the pin count of a dirty metadata buffer is stored in its associated log item.

All log items that have associated "dirty but logged" metadata buffers are linked into a global *sync list* in ascending LSN order. The LSN of the first log item on the sync list indicates that the log space at and after the LSN still contains active log data. The only way to reclaim log space is to write the metadata buffers associated with the log items in place. This can be done by the log daemon if need be. After a metadata buffer is written in place, its log item can be removed from the sync list and freed.

Because we do not need to record the whole history of metadata updates, the on-disk log area can be used in a circular fashion. The log area can be either internal (in this case, we allocate log area from the middle AG to reduce disk head motion) or external. At the beginning of the log area there are two copies of *log anchors*. The log anchors are checksumed, timestamped, and updated alternately. They, in conjunction with the log area information stored in the superblock, record the current tail of the active log area.

To avoid starting a transaction needlessly, all sanity checks must be done beforehand. Furthermore, we must reserve three kinds of resources (disk space, locked buffer, and log space) to make sure that a transaction runs to completion safely. The disk space reservation is for the maximum number of disk blocks that will be requested by a transaction. The locked buffer reservation is for the maximum number of metadata buffers that will be locked by a transaction. This reservation guarantees that a transaction can continue to grab new buffers within its declared requirement. To prevent yFS from taking over all the buffers in the system, the total number of buffers locked and pinned by yFS cannot exceed a certain threshold at any given time (Section 5). The log space reservation avoids a deadlock situation that happens when the log space is low but the first log item on the sync list is locked by the committing transaction.

For example, the transaction CREATE_DINODE is used to create a chunk of disk inodes in a given AG. Let us suppose that a chunk of disk inodes contains 64 disk inodes, each inode is 512 bytes, the block size is 16 KB, and the maximum height of a B*-tree is 5. In this case, we need to reserve 7 blocks of disk space (5 blocks for each level of the B*-tree and two blocks for the inodes themselves). In addition, the transaction will modify the AG bitmap block and the AG inode block. The maximum number of buffers that can be locked by this transaction is 9 (7, as indicated above plus two more blocks—one for the

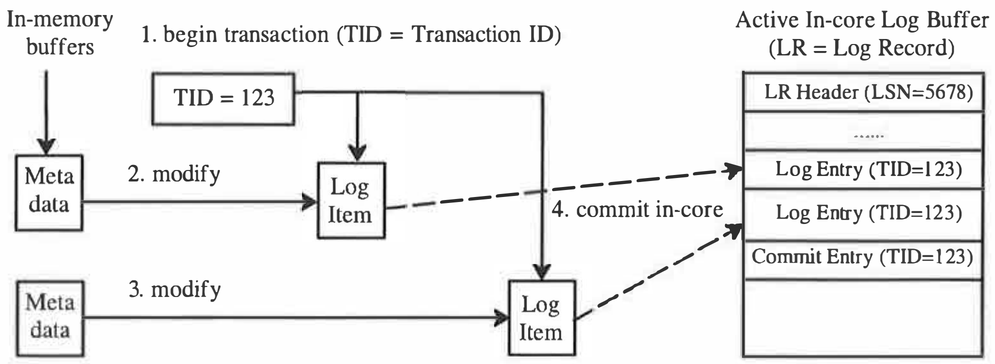


Figure 2. Data Structures Involved in a Transaction

bitmap block and another for inode block containing the root of the B*-tree and other status information). The log space needed per block is estimated to be half the block size, which is usually more than enough due to our use of fine-granularity logging. So the log space to be reserved is 73,728 bytes plus a little overhead for log entries and log record headers. Because reservations are made against the maximum possible requirements, any unused resource must be returned when the transaction commits in core.

4.4 Beyond Transactions

The transaction model described above provides us with a foundation to move the file system from one consistent state to another safely. However, we still need some extra mechanisms to deal with three special cases described below.

Inode Recovery List. POSIX semantics require that a file whose link count drops to zero continues to exist until the last reference to it is removed. This feature is often used to create temporary files. However, it poses a serious problem to journaling code because we do not know when the file will be truncated. Therefore we cannot decrement its link count and delete the file in one atomic transaction. The solution is to put such an inode on an AG Inode Recovery List (IRL) when its link count is decremented to zero. When the reference count of that file becomes zero, we truncate the file and remove the disk inode from the IRL with one transaction. If there is a crash after a file's link count drops to zero but before it is truncated, the disk inode will be found on an IRL during recovery and the intended truncation can be performed.

Extent Log Item. Locking is an important requirement in a journaling file system. To avoid deadlock, a transaction can choose to allocate disk blocks from only one AG. However, we cannot have this kind of control during deallocation because blocks to be freed could have been allocated from more than one AG. Our solution is to use more than one transaction to accomplish the job. The first transa tion frees extents from the corresponding file's block map and logs the fact that these extents are to be freed from the file system bitmap using a "marked free" extent log item. After this transaction has committed on the disk, a *bitmap daemon* will start transactions to free the marked extents from the bitmap and write corresponding "unmark free" extent log items. By doing this, we not only avoid deadlocking, but also prevent any freed metadata block from being reused prematurely. Note that if a freed block was used for metadata, the first transaction has to write a special log entry indicating that the metadata block was deleted. These special log entries will be used in crash recovery (Section 4.5).

Dependent Data Block List. This last mechanism is used to make sure that a newly created file contains useful data and no file data is lost due to reallocation, making use of the concept of "dependent data blocks." Every buffer that contains newly allocated disk space is added to the dependent data block list of the ongoing transaction. These buffers are later moved over to the dependent data block list of the incore log buffer that contains the commit log entry of the transaction. Before an incore log buffer is flushed, any data buffers on its dependent data block list must be written first. This helps in enforcing the three update dependencies proposed by McKusick and Ganger in [16].

4.5 Log Recovery

Log recovery is relatively easy because of the use of physical logging. First, we read the latest valid copy of the log anchor to find the tail of the log. Then we scan forward to find all the metadata blocks that were deleted. Note that whenever we delete a metadata block, we have to write its block number into the log. If this was not done, we could end up overwriting valid user data by replaying obsolete metadata updates. This analysis scan stops at the head of the log, which is defined as the valid log record with the highest LSN. During the second scan from tail to head, we redo any changes to the metadata blocks, ignoring any stale metadata blocks. If we encounter a "marked free" extent log item that does not have a corresponding "unmark free" extent log item, we free such extents in the corresponding bitmap.

The recovery of inodes is done by the kernel after the file system is mounted. At this point, the log area is completely empty and we can use the normal transaction mechanisms to truncate any inodes on an IRL.

5 Implementation

Our prototype has been implemented as a kernel loadable module using the stackable vnode interface on the FreeBSD 4.5-Release. Several user-level utilities have also been developed to create a file system, debug a file system, and collect run-time statistics. The total source code consists of about 30,000 lines of C code. For brevity, we discuss only the key implementation details.

Our first critical decision concerned the encapsulation of metadata updates into transactions. If we follow the transaction model closely, we have to lock all metadata—including vnodes—needed to perform a transaction within its context. However, after studying the VFS layer carefully, we found out that vnode locking and unlocking are normally done within the VFS layer. Therefore, we decided to leave vnode locks outside a transaction's context. All changes to an inode are copied to its corresponding inode block when a transaction commits. This arrangement has two big benefits: (1) It keeps all transaction code within yFS while leaving the VFS layer inta t; this is helpful in achieving our goal to be as unobtrusive as possible. (2) Because vnodes are managed by the VFS layer, leaving them out of the transaction context makes it possible for the VFS layer to reclaim vnodes freely.

Our second challenge was to find a way to represent metadata blocks. Because we use B*-trees and implement a directory directly on disk blocks instead of on the top of a regular file, we can no longer identify individual metadata blocks using negative block numbers as FFS does [4]. Therefore, all metadata blocks are attached to the device vnode corresponding to the disk device of the file system.

Since buffers of the device vnode use VM pages (i.e., page cache), metadata are cached longer than the lifetime of buffers that were used to access them. However, using the device vnode leads to the problem of identifying the metadata buffers that belong to each individual file. This information is needed for the *fsync()* operation. To solve this problem, we record in the inode the LSN of the transaction that made the latest changes to its file. To flush the metadata of a file, we only have to the make sure that the incore log buffer tagged with that LSN is flushed.

Our third challenge arose from the need to pin dirty metadata buffers until corresponding transactions have committed on disk. Fortunately, FreeBSD has a B_LOCKED queue that locks a buffer in the memory. All we have to do is to keep the buffer on this queue as long as its pin count is non-zero. In addition, a few fields need to be added to the buffer header to support transaction semantics. The maximum number of buffers that can be pinned down by yFS simultaneously is limited to be one half of all the available buffers in the system by default.

Our last challenge was keeping logging overhead low. Besides using fine-granularity logging, we take some additional measures to reduce the logging overhead:

(1) Log changes are made when absolutely necessary. For example, the bitmap encoding information and the summary tree do not have to be logged because these derived data can be reconstructed easily from the underlying bitmaps.

(2) Optimized data structures are used to reduce the logging I/O. An example of this is the maintenance of the directory entries in a directory block as a linked list ordered by hash values. As a result of this, the creation of an entry requires only the new entry itself and the modified pointer(s) to be logged. The deletion of an old entry requires only one modified pointer to be logged to remove the entry from the list.

Reducing logging information saves memory copies and I/O involved in logging. It also allows dirty metadata blocks to remain cached longer without being flushed because log space is consumed more slowly.

6 Experimental Evaluation

In this section we evaluate the performance of yFS against FFS enhanced with Soft Updates (FFS-SU). Our experimental comparisons are restricted to FreeBSD file systems for the following reasons. First, the effort needed to port non-FreeBSD file systems to FreeBSD for the sake of comparison against yFS is well beyond the scope of this work. Second, the comparison would not be fair because file system implementations are intimately tied to the virtual memory, scheduling, and I/O subsystems. Consequently, it would be difficult to identify what causes the performance differences between yFS and these ported file systems. Third, although the original FFS was designed almost two decades ago, it has benefited from two major improvements: clustered I/O [14] and Soft Updates [15,16,17]. The FreeBSD implementation of FFS further optimizes FFS with a variety of techniques [19]. In a nutshell, FFS-SU is a formidable file system against which to compare yFS.

The platform used in our measurements has an Intel Xeon 500 MHz. CPU, 128 Mbytes memory, an Adaptec AIC-7890 SCSI Adapter, and two 9.1 GB 7,200-RPM Seagate ST39140 SCSI disks. The first disk is used as the operating system disk. The second disk is used as the file system disk, where we created our test file systems.

All file systems are formatted with a block size of 16 KB and a fragment size of 2 KB (both are defaults under FreeBSD 4.5). The average file size parameter is 16 KB and the average number of files per directory parameter is 64. The allocation group (called cylinder group in FFS) size is 178 Mbytes for all file systems. For yFS, the log area is 32MB, which can be either inline (yFS-inline) or on a separate device (yFS-external). Because we have only two disks, the external log is created *on the operating system disk* in the case of yFS-external.

The four benchmarks used in this section are kernel build, PostMark, archive extraction, and file system aging. We start each benchmark run with a cold cache. All results are averaged over at least five runs; the standard deviation is usually less than 3% of the average, with a value as high as 6% in the aging benchmark.

We have also instrumented the kernel device driver to collect two kinds of low-level I/O statistics *on the file system disk only*: total number of I/O requests and average I/O times in milliseconds. The total number of I/Os reported (as X+Y) has two parts: the number of I/Os performed during the lifetime of a benchmark plus the number of I/Os performed after the benchmark terminates (e.g. FFS-SU issues additional I/Os for background deletion after PostMark finishes). The I/O times reported (as W+Z) also have two components: the service time (W, the time between the initiation of an I/O to the disk controller and the receipt of the corresponding interrupt) and the driver-level queuing time (Z). Note that the I/O times reported are only for the lifetime of each benchmark.

We ran all the benchmarks on FFS without Soft Updates as well. However, this version of FFS supports different semantics than both yFS and FFS-SU as all metadata operations are performed synchronously, instead of being performed asynchronously. These synchronous I/Os completely dominate the performance and make the FFS numbers uninteresting as points of comparison.

	Copy Phase Time (s)	Compile Phase Time (s)	Total I/O Requests	Avg I/O Times (ms)
yFS-inline	32	704	10505+131	5+58
yFS-external	32	703	10177+115	4+53
FFS-SU	40	701	14614+181	7+209

Table 1 Kernel build benchmark user-level and low-level I/O results

6.1 Kernel Build Benchmark

This benchmark uses a small shell script to copy kernel files from the operating system disk to the test file system created on the file system disk. It then builds the FreeBSD generic kernel and all kernel modules under the test file system. The copy phase of this benchmark is I/O intensive. During this phase, the benchmark copies files from */usr/src/sys* on the operating system disk to the test file system. The compile phase is CPU bound.

As shown in Table 1, the two yFS configurations perform 20% better than FFS-SU in the copy phase (32 seconds vs. 40 seconds). In FFS-SU, successively created small files may not be allocated contiguously to each other because a partial block cannot span across a block boundary. In addition, the last partial block of a file may not be allocated close to its preceding block [28]. In contrast, yFS is able to allocate small files adjacent to each other because it does not enforce artificial block boundaries. Since yFS does not divide disk space into blocks statically, it cannot maintain a free block count. Therefore, the directory layout algorithm described in Section 3.3 uses a free fragment count as the metric of free space. This tends to result in more locality in accessing, as compared to FFS-SU, where free space provided by fragments is ignored by the algorithm. These features certainly help yFS in other benchmarks as well.

Both yFS configurations perform quite comparably to FFS-SU in the compile phase even though yFS uses more complicated algorithms than FFS-SU. FFS-SU generates more I/Os than yFS. But this does not hurt the apparent performance of FFS-SU due to the overlap between I/O and CPU processing.

6.2 PostMark Benchmark

PostMark is a popular benchmark that simulates the working environment of a Web/News server [26]. It creates an initial pool of specified number of files. Then it performs a mix of creation, deletion, read, and append operations. Finally, all files are deleted. We use default configurations of PostMark v1.5 (the size range is between 500 bytes and 9.77 KB, both read and write block sizes are 512 bytes, the read/append and create/delete ratios are 5) except the base number of files. The results are shown in Tables 2(a) and 2(b). There are two interesting points to make at this point.

First, unlike other benchmarks used in this paper, FFS-SU performs fewer I/Os than yFS (the numbers for yFS-external do not include logging I/O). This is due to the following two reasons: (1) yFS needs four times as many I/Os as FFS-SU to write disk inodes because its inodes are four times larger than those of FFS-SU. (2) FreeBSD has a fast path deletion optimization [3], which deletes newly created files immediately if the inodes have not yet been written. yFS has to faithfully write log information for every metadata update, even if the update turns out to be cancelled before it reaches disk. Note that the average service time per I/O for FFS-SU is longer than that of yFS.

Second, the deletion rate (number of files deleted per

	50,000 Files		100,000 Files		150,000 Files	
	Total Time (s)	Deletion Rate	Total Time (s)	Deletion Rate	Total Time (s)	Deletion Rate
yFS-inline	102	1666	215	1552	320	1530
yFS-external	87	2772	177	2795	267	2560
FFS-SU	130	6962	281	5888	462	5357

Table 2(a) PostMark (v1.5) benchmark user-level results

	50,000 Files		100,000 Files		150,000 Files	
	Total I/O Requests	Avg I/O Times (ms)	Total I/O Requests	Avg I/O Time (ms)	Total I/O Requests	Avg I/O Times (ms)
yFS-inline	58084+373	2+39	117247+407	2+34	176159+358	2+33
yFS-external	56874+402	1+28	114698+462	1+24	172104+482	1+23
FFS-SU	57632+404	4+107	115603+575	4+100	173222+397	4+98

Table 2(b) PostMark (v1.5) benchmark low-level I/O results seconds) of FFS-SU is much higher than those of the two yFS configurations. This can be partly attributed to the fast path deletion optimization and the background deletion employed by FFS-SU. For example, the *df –i* command shows 22,471 inodes still in use in the case of 100,000 base files for FFS-SU after the benchmark finishes.

PostMark is the most metadata intensive test of the four benchmarks used in this paper. Let us consider the statistics for FFU-SU and yFS-internal for the 150,000 base file case. For Soft Updates, the benchmark process has to push the work item list 1,209 times itself to help the syncer daemon and sleep 7 times to slow it down. In the meantime, 13,218 metadata buffers are re-dirtied due to rollbacks. For yFS-internal, the small 32 MB log area wraps around 10 times. The benchmark process waits 108 times for incore log buffers and 10 times for log space. In comparison, yFS-inline fares better than FFS-SU by 31% (320 seconds vs. 462 seconds). Because other features do not help yFS here (e.g., yFS cannot inline files larger than 420 bytes), this benchmark clearly shows the efficiency of our journaling scheme.

PostMark writes all test files in the same directory (i.e., the root) by default, creating a large directory. The non-linear directory organization used by yFS can efficiently deal with this. FFS-SU also copes with the large directory by using its directory hashing scheme [19].

6.3 Archive Extraction Benchmark

Archive extraction is a common pre-requisite of installing a software package on a Unix-like system. This benchmark extracts the file *ports.tgz* (15,165,655 bytes, consisting of 6,470 FreeBSD 4.5 ports) with the *tar* command and then deletes these files with the *rm* command. We un-mount and re-mount the file system between the two operations to remove any impact of caching. The results of this benchmark are shown in Table 3. Both yFS configurations win in this benchmark. For example, yFS-external beats FFS-SU by 82% (89 seconds vs. 502 seconds) in the creation phase and 75% (37 seconds vs. 148 seconds) in the deletion phase.

This benchmark creates a total of 55,189 files. Out of 10,448 directory files (including the root), 10,257 are stored in the inline format in yFS. Out of 44,742 regular files, 24,368 are stored in the inline format in yFS. For all of these inline files in yFS, FFS-SU has to allocate disk space for them separately from their disk inodes. This is a major reason for the noticeable performance gap between yFS and FFS-SU. In addition, FFS-SU re-dirties 35,393 buffers due to its rollback operations, again handicapping FFS-SU against yFS. Note that deleting an inline file does not need to update disk space bitmap.

In the deletion phase, yFS even beats FFS-SU featuring background deletion. In FFS-SU, freeing a disk inode incurs an extra I/O to zero its mode field to help *fsck* identify unused inodes [12]. yFS does not do this because of its strong atomic guarantees. Furthermore, for all 55,189 files, yFS requires 1,725 I/Os to read their big 512-byte disk inodes compared to 432 I/Os needed by FFS-SU to read the smaller 128-byte inodes. The benefit is that yFS saves 10,257 I/Os that are required by FFS-SU to read the small directories.

6.4 File System Aging Benchmark

File system aging has been used to demonstrate the effectiveness of several layout optimizations in FFS [9,28]. While it is important to understand the long-term effects of the yFS allocation policy, that work is beyond the scope of this paper. The results of this test are thus not indicative of the long-term behavior of yFS.

Our file system aging benchmark performs a mix of file system operations to fill an empty file system with directories and regular files. The inclusion of directories is essential because they play an important role in file system layout. Each operation must have a working directory. Initially, there is only one root directory so the working directory is the root. As more directories are created, a working directory is selected randomly among them with a uniform distribution. Each file operation could be either a creation or a deletion. The probability that the next operation is a creation decreases gradually as the file system fills up. If we choose to create, the probability of creating a directory is 1/N (N defaults to 64), assuming that there are N files per directory on the average. For the file sizes, 93% are determined by the Lognormal distribution (μ=9.357, σ=1.318) and the rest 7% are decided by the Pareto distribution (κ=133K, α=1.1) [27]. If we choose to delete, we first order the files in the directory alphabetically before picking a victim. This guarantees that we delete the same file even if the on-disk directory structures are different for different file systems. All file names are created randomly from a set of

	Creation Phase			Deletion Phase		
	Elapsed Time (seconds)	Total I/O Requests	Avg I/O Times (ms)	Elapsed Time (seconds)	Total I/O Requests	Avg I/O Times (ms)
yFS-inline	97	26472+558	4+75	43	4683+141	12+111
yFS-external	89	25761+647	4+59	37	3929+238	11+107
FFS-SU	502	96614+201	10+645	148	22613+303	10+185

Table 3 Archive extraction benchmark user-level and low-level I/O results

characters. The results of this benchmark are shown in Tables 4(a) and 4(b).

Both yFS configurations excel in this benchmark. Since this benchmark performs a variety of operations, the performance gains for yFS come from many of the reasons mentioned for the other tests. In particular, the rollback operations in FFS-SU take a toll on its performance. Note that the performance gap grows wider as the number of operations grows (yFS-external beats FFS-SU by 30%, 40%, and 49% respectively). This shows that the metadata handling capability of yFS scales up better than that of FFS-SU. When the memory holds too many dirty metadata buffers that are constrained (by WAL or pending dependencies), large sequential I/Os, as used in yFS, offer a better way than delayed individual writes (as used in FFS-SU) to clear the backlog.

An unexpected phenomenon about this benchmark is that sometimes yFS–inline outperforms yFS-external. A closer look at the directory layout algorithm reveals that the AG of a directory *directly* under the root is chosen *randomly*. Because the working directory for each operation is also chosen randomly in this benchmark, the distances between these top-level directories can play a prominent role in determining the overall performance.

6.5 Discussions of the Results

Generally, yFS-external has an advantage over yFS-internal because it removes logging I/O from the normal I/O path. However, if a benchmark is not metadata intensive (Section 6.1) or has some other quirks (Section 6.4, for example), using a separate logging disk does not guarantee an improvement in apparent performance. Nonetheless, yFS outperforms FFS-SU on all our benchmarks except the kernel compile phase.

One of the major reasons for the performance gains of yFS is the use of a lightweight journaling scheme. Each transactions writes less than 500 bytes of log data, incurring a small overhead.

yFS uses extent-based allocation to allocate disk space for files and B*-trees to represent the extents of a file. Because we do not use large files (the largest file used in our aging benchmark is 52,242,324 bytes) and we run all benchmarks on an empty file system, the benefits gained by these features are limited. Although extent-based allocation and B*-tree algorithm should make yFS handle large data sets efficiently as other file systems with similar algorithms have claimed [6,7], it is difficult to demonstrate this advantage without proper aging on an empty file system. That work is in progress and beyond the scope of this paper.

7 Related Work

The yFS disk space management algorithm is an adaptation of the JFS scheme [6]. However, the original JFS scheme uses one allocation tree for the entire file system and two bitmaps (working and permanent) to track disk usage. We do neither of these because of their potential problem in terms of concurrency and space overhead. JFS does not use the location information inherent within the summary tree to search the tree. In JFS, when a right buddy is merged with its left buddy, its value is set to be NOFREE, disregarding the fact that the portion of the bitmap covered by this node may actually contain a free extent. As a result, JFS searches the summary tree using only the first-fit algorithm.

XFS uses dual B*-trees to track free extents by block number and by extent size respectively [7,8]. Each allocation and deallocation of disk space must update both trees. Since the two B*-trees grow and shrink on the fly, their blocks are not guaranteed to be stored contiguously. Although XFS only records free extent information, the amount of disk space used by the space management routine itself is not necessarily less than that for the bitmap solution when free disk space becomes fragmented.

Unlike JFS and XFS, yFS inherits the tradition of FFS to use fragments as the basic allocation unit. Large files (defaulting to more than 12 blocks) are allocated in full blocks. Although saving disk space is a consideration, the main motivation in yFS is to have the ability to cluster small files closer to gain better performance. Unlike FFS,

Elapsed Time	10,000 Operations (seconds)	20,000 Operations (seconds)	40,000 Operations (seconds)
yFS-inline	114	241	552
yFS-external	117	241	519
FFS-SU	168	400	1024

Table 4(a) File system aging benchmark user-level results

	10,000 Operations		20,000 Operations		40,000 Operations	
	Total I/O Requests	Average I/O Times (ms)	Total I/O Requests	Average I/O Times (ms)	Total I/O Requests	Average I/O Times (ms)
yFS-inline	20961+325	10+144	40967+454	11+150	82860+610	12+165
yFS-external	20521+216	11+148	40714+414	11+147	80034+673	12+155
FFS-SU	21817+387	15+240	47791+365	16+281	121179+452	16+335

Table 4(b) File system aging benchmark low-level I/O results we do not maintain a separate bitmap for cluster allocation because the buddy encoding and summary tree do a good job in speeding up the bitmap lookup. As a result, the number of fragments per block is not limited to eight, making the trade-off between disk space and throughput less painful.

One important difference between FFS and yFS is that a full or partial block can start at *any* fragment address in yFS. Therefore, yFS uses a free fragment count instead of a free block count to keep track of the amount of free space. In FFS, the free space provided by fragmented blocks is not considered by the directory layout algorithm, resulting in lower locality. Because there are no hard block boundaries in yFS, the external fragmentation between files is also reduced.

The use of extent maps has been suggested recently as one of the improvements planned for Ext2/3 by Ts'o and Tweedie [29]. The preferred form of this improvement avoids the use of B*-trees for storing extent maps and stores extent information inline within an inode or into a single block. Traditional indirect blocks are used when the inline and the single block storage is not enough. Fragmentation, which forces the system to revert to the old scheme, is avoided using preallocation. Performance considerations make preallocation almost mandatory in the suggested improvement. yFS implements extent maps in a different manner, allowing for three different inode formats, including one that stores *data* inline. yFS also implements a full-fledged B*-tree algorithm that is invoked only when necessary instead of altogether avoiding its use.

Our transaction model is similar to that of XFS [7,8] but with some notable differences. First, we reserve locked buffer resources before a transaction starts to avoid deadlocks. Although this is an issue that must be addressed, XFS does not appear to do so. To do this correctly, the file system must be able to detect the memory pressure. Second, each transaction in yFS only logs metadata modified by itself. XFS uses *accumulated logging* and a fixed 128 byte logging granularity allowing for the reclamation of log space without writing metadata in place. Our experiments show that the amount of logging I/O increases greatly because modifications made by previous transaction(s) to the same metadata are relogged *repeatedly*. Third, we use two queues to maintain incore log buffers instead of a closed circular queue in XFS. This makes it easy to dynamically adjust the number of incore log buffers to match the metadata update rates. Lastly, we do not lock inodes in a transaction's context. Any changes to a disk inode are copied to its corresponding inode block buffer when a transaction commits in core.

Seltzer et al. compare the journaling and Soft Updates techniques and conclude that they are largely comparable [17]. However, Soft Updates requires a non-trivial amount of memory to maintain metadata dependency at fine granularity. Although it does not enforce an order on buffer writes, it does introduce rollback and roll forward operations that increase memory and I/O overhead. For Soft Updates to work, it must have intimate knowledge of the inter-relationship between metadata that could be involved in a single operation. This dependency tracking could become tricky, if not entirely impossible, to work on complex data structures such as B*-trees, where the pointers can move within a metadata block. While Soft Updates has the ability to delay metadata updates, the memory tends to fill up. When that happens, the only way to get rid of the accumulated dependencies is to resort to synchronous writes. In case of a crash, a background *fsck* run is still needed to salvage any unused resources and both old and new names can show up due to an interrupted rename operation.

Instead of starting a transaction to accept metadata changes made by each atomic operation, Ext3 simply creates one compound transaction once in a while to receive all the changes made after a previous transaction closes [5]. This simple transaction design does not support fine-granularity logging. As a result, a modified metadata block is logged in its entirety no matter how minute the modification is. If a new transaction wants to modify a buffer that is being flushed by a committing transaction, a memory copy must be performed.

LFS is a write-oriented file system that logs both file data and metadata [13,14]. Its most glaring problem is its cleaning overhead. Although some techniques have been proposed to reduce cleaning overhead, their effectiveness depend on factors like workload, idle time, and access patterns [30,31,32]. Furthermore, LFS inherits some of the old data structures used by FFS such as the highly skewed addressing tree for each file. The maximum extent length is also limited by the size of a segment, which cannot be made large due to cleaning considerations. yFS does not have any of these problems.

8 Conclusions

This paper describes the design and implementation of yFS, a journaling file system for FreeBSD that represents a synthesis of existing and new ideas for improving file system performance. Our experimental results show that yFS works better than Soft Updates even without a dedicated logging device. For the benchmarks we have investigated, yFS's performance edge can be attributed to the use of lightweight logging, inline data storage, and the relaxation of the usual block boundary constraints.

9 Acknowledgements

We would like to thank our shepherd, Prof. Margo Seltzer, and the anonymous reviewers for their comments, which were useful in improving this paper.

10 References

[1] Edward Grochowski and Roger F. Hoyt. Future Trends in Hard Disk Drives. IEEE Transactions on Magnetics, pp. 1850-1854, Vol. 32, May 1996.

[2] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. USENIX Annual Technical Conference, pp. 41-54, June 2000.

[3] The FreeBSD open source project. http://www.freebsd.org.

[4] Marshall Kirk McKusick, et al. A Fast File System for UNIX. ACM Transaction on Computer Systems, Vol. 2, No. 3, pp. 181-197, August 1984.

[5] Stephen C. Tweedie. Journaling the Linux ext2fs Filesystem. LinuxExpo'98, May 1998.

[6] The IBM JFS open source project. http://oss.software.ibm.com/developerworks/opensource/jfs/.

[7] The SGI XFS open source project. http://oss.sgi.com/projects/xfs/.

[8] Adam Sweeney, et al. Scalability in the XFS File System. USENIX Annual Technical Conference, pp. 1-14, January 1996.

[9] Keith A. Smith and Margo I. Seltzer. A Comparison of FFS Disk Allocation Policies. In USENIX Annual Technical Conference, pp. 15-26, January 1996.

[10] Eric H. Herrin II and Raphael A. Finkel. The Viva File System. Technical Report No. 225-93. University of Kentucky, Lexington, 1993.

[11] Shankar Pasupathy. Implementing Viva on Linux. http://www.cs.wisc.edu/~shankar/Viva/viva.html, July 1996.

[12] Marshall Kirk McKusick. Fsck—The Unix File System Check Program. Computer Systems Research Group, UC Berkeley, 1985.

[13] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, Vol. 10, No. 1, pp. 26-52, February 1992.

[14] Margo I. Seltzer, Keith Bostic, et al. An Implementation of a Log-Structured File System for UNIX. Proceedings of the 1993 Winter USENIX Conference, pp. 307-326, January 1993.

[15] Gregory R. Ganger, Yale N. Patt. Metadata Update Performance in File Systems. USENIX Symposium on Operating Systems Design and Implementation, pp. 49-60, November 1994.

[16] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem, USENIX Annual Technical Conference, FREENIX Track, pp. 1-17, June 1999.

[17] Margo I. Seltzer, et al. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. pp. 71-84, USENIX Annual Technical Conference, June 2000.

[18] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. USENIX Annual Technical Conference, pp. 77-89, June 1998.

[19] Ian Dowse and David Malone. Recent Filesystem Optimisations on FreeBSD. USENIX Annual Technical Conference, FREENIX Track, pp. 245–258, June 2002.

[20] Veritas File System white papers. Available at: http://www.intel–sol.com/products/veritas/.

[21] Michael J. Folk, Bill Zoellick, and Greg Riccardi. File Structures: An Object-Oriented Approach with C++, 3nd Edition. Addison-Wesley, 1998.

[22] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, Inc., 1993.

[23] Daniel Phillips. A Directory Index for Ext2. http://people.nl.linux.org/~phillips/htree/paper/htree.html, September 2001.

[24] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System, 16th ACM Symposium on Operating Systems Principles, pp. 224-237, October 1997.

[25] Kenneth W. Preslan, et al. A 64-bit Shared Disk File System for Linux. Sixteenth IEEE Mass Storage Systems Symposium, March 1999.

[26] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR3022. Network Appliance Inc., October 1997.

[27] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. Proceedings of the ACM SIGMETRICS, pp. 151-160, June 1998.

[28] Keith A. Smith and Margo I. Seltzer. File System Aging—Increasing the Relevance of File System Benchmarks. Proceedings of the ACM SIGMETRICS, pp. 203-213, June 1997.

[29] Theodore Ts'o and Stephen Tweedie. Planned Extensions to the Linux Ext2/Ext3 Filesystem. USENIX Annual Technical Conference, FREENIX track, pp. 235-244, June 2002.

[30] Trevor Blackwell, Jeffrey Harris, Margo I. Seltzer, Heuristic Cleaning Algorithms in Log-Structured File Systems, USENIX Technical Conference, pp. 277-288, January 1995.

[31] Jeanne Neefe Matthews, et al. Improving the Performance of Log-Structured File Systems with Adaptive Methods. Sixteenth ACM Symposium on Operating System Principles, pp. 238-251, October 1997.

[32] Jun Wang and Yiming Hu, WOLF—A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. 1st Conference on File and Storage Technologies, pp. 47-60, January 2002.

Semantically-Smart Disk Systems

Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison

Abstract

We propose and evaluate the concept of a semantically-smart disk system (SDS). As opposed to a traditional "smart" disk, an SDS has detailed knowledge of how the file system above is using the disk system, including information about the on-disk data structures of the file system. An SDS exploits this knowledge to transparently improve performance or enhance functionality beneath a standard block read/write interface. To automatically acquire this knowledge, we introduce a tool (EOF) that can discover file-system structure for certain types of file systems, and then show how an SDS can exploit this knowledge on-line to understand file-system behavior. We quantify the space and time overheads that are common in an SDS, showing that they are not excessive. We then study the issues surrounding SDS construction by designing and implementing a number of prototypes as case studies; each case study exploits knowledge of some aspect of the file system to implement powerful functionality beneath the standard SCSI interface. Overall, we find that a surprising amount of functionality can be embedded within an SDS, hinting at a future where disk manufacturers can compete on enhanced functionality and not simply cost-per-byte and performance.

1 Introduction

"To know that we know what we know, and
that we do not know what we do not know,
that is true knowledge." *Confucius*

As microprocessors and memory chips become smaller, faster, and cheaper, embedding processing and memory in peripheral devices has become an increasingly attractive proposition [1, 19, 32, 40]. Placing processing power and memory capacity within a "smart" disk system allows functionality to be migrated from the file system into the disk (or RAID), thus providing a number of potential advantages over a traditional system. For example, when computation takes place near data, one can improve performance by reducing traffic between the host processor and disk [1]. Further, such a disk system has and can exploit low-level information not typically available at the file-system level, including exact head position and block-mapping information [26, 35]. Finally, unmodified file systems can leverage these optimizations, enabling deployment across a broad range of systems.

Unfortunately, while smart disk systems have great promise, realizing their full potential has proven difficult. One causative reason for this shortfall is the *narrow interface* between file systems and disks [16]; the disk subsystem receives a series of block read and write requests that have no inherent meaning, and the data structures of the file system (*e.g.*, bitmaps for tracking free space, inodes, data blocks, directories, and indirect blocks) are not exposed. Thus, research efforts have been limited to applying disk-system intelligence in a manner that is oblivious to the nature and meaning of file system traffic, *e.g.*, improving write performance by writing blocks to the closest free space on disk [15, 40].

To fulfill their potential and retain their utility, smart disk systems must become "smarter" while the interface to storage remains the same. Such a system must acquire knowledge of how the file system is using it, and exploit that understanding in order to enhance functionality or increase performance. For example, if the storage system understands which blocks constitute a particular file, it can perform intelligent prefetching on a per-file basis; if a storage system knows which blocks are currently unused by the file system, it can utilize that space for additional copies of blocks, for improved performance or reliability. We name a storage system that has detailed knowledge of file system structures and policies a *Semantically-Smart Disk System (SDS)*, since it understands the meaning of the operations enacted upon it.

An important problem that must be solved by an SDS is that of "information discovery" – how does the disk learn about the details of file system on-disk data structures? The most straight-forward approach is to assume the disk has exact "white-box" knowledge of the file system structures (*e.g.*, access to all relevant header files). However, in some cases such information will be unavailable or cumbersome to maintain. Thus, in this paper, we explore a "gray-box" approach [4], attempting where possible to automatically obtain such file-system specific knowledge within the storage system.

We develop and present a fingerprinting tool, EOF, that automatically discovers file-system layout through probes and observations. We show that by using EOF, a smart disk system can automatically discover the layout of a certain class of file systems, namely those that are similar to the Berkeley Fast File System (FFS) [27].

We then show how to exploit layout information to infer higher-level file-system behavior. The processes of *classification, association,* and *operation inferencing* refer to the ability to categorize each disk block (data, inode, bitmaps, or superblock) and detect the precise type of each data block (file, directory, or indirect pointer), to associate each data block with its inode or other relevant information, and to identify higher-level operations such as file creation and deletion. An SDS can use some or all of these techniques to implement its desired functionality.

To prototype a smart disk system, we use a software infrastructure in which an in-kernel driver interposes on read and write requests between the file system and the disk. In our prototype environment, we can explore most of the challenges of adding functionality within an SDS, while adhering to existing interfaces and running underneath a stock file system. In this paper, we focus on the Linux ext2 and ext3 file systems, as well as NetBSD FFS.

To understand the performance characteristics of an SDS, we study the overheads involved with fingerprinting, classification, association, and operation inferencing. Through microbenchmarks, we quantify costs in terms of both space and time, demonstrating that common overheads are not excessive.

Finally, to illustrate the potential of semantically-smart storage systems, we have implemented a number of case studies within our SDS framework: aligning files with track boundaries to increase the performance of small-file operations [35], using information about file-system structures to implement more effective second-level caching schemes with both volatile and non-volatile memory [43], a secure-deleting disk system that ensures non-recoverability of deleted files [20], and journaling within the storage system itself to improve crash recovery time [21]. Through these case studies, we demonstrate that a broad range of functionality can be implemented within a semantically-smart disk system. In some cases, we also demonstrate how an SDS can tolerate imperfect information about the file system, which is a key to building robust semantically-smart disk systems.

The rest of this paper is organized as follows. In Section 2, we discuss related work. We then discuss file-system fingerprinting in Section 3, classification and association in Section 4, and operation inferencing in Section 5. We evaluate our system in Section 6, and present case studies in Section 7. We conclude in Section 8.

2 Related Work

The related work on smart disks can be grouped into three categories. The first group assumes that the interface between file and storage systems is fixed and cannot be changed, the category under which an SDS belongs. Research in the second group proposes changes to the storage interface, requiring that file systems be modified to leverage this new interface. Finally, the third group proposes changes not only to the interface, but to the programming model for applications.

Fixed interfaces: The focus of this paper is on the integration of smart disks into a traditional file system environment. In this environment, the file system has a narrow, SCSI-like interface to storage, and uses the disk as a persistent store for its data structures. An early example of a smart disk controller is Loge [15], which harnessed its processing capabilities to improve performance by writing blocks near the current disk-head position. Wang *et al.*'s log-based programmable disk [40] extended this approach in a number of ways, namely quick crash-recovery and free-space compaction. Neither of these systems assume or require any knowledge of file system structures.

When storage system interfaces are more developed than that provided in the local setting, there are more opportunities for new functionality. The use of a network packet filter within the Slice virtual file service [3] allows Slice to interpose on NFS traffic in clients, and thus implement a range of optimizations (*e.g.*, preferential treatment of small files). Interposing on an NFS traffic stream is simpler than doing so on a SCSI-disk block stream because the contents of NFS packets are well-defined.

High-end RAID products are the perfect place for semantic smartness, because a typical enterprise storage system has substantial processing capabilities and memory capacity. For example, an EMC Symmetrix server contains up to eighty 333 MHz Motorola microprocessors and can be configured with up to 64 GB of memory [14]. Some high-end RAID systems currently leverage their resources to perform a bare minimum of semantically-smart behavior; for example, storage systems from EMC can recognize an Oracle data block and provide an extra checksum to assure that a block write (comprised of multiple sector writes) reaches disk atomically [7]. In this paper, we explore the acquisition and exploitation of more detailed knowledge of file system behavior.

More expressive interfaces: Given that one of the primary factors that limits the addition of new functionality in a smart disk is the narrow interface between file systems and storage, it is not surprising that there has been research that investigates changing this interface. We briefly highlight these projects. Mime investigates an enhanced interface in the context of an intelligent RAID controller [9]; specifically, Mime adds primitives to allow clients to control both when updates to storage become visible to other traffic streams and the commit order of operations. Logical disks expand the interface by allowing the file system to express grouping preferences with lists [11]; thus, file systems are simplified since they do not need to maintain this information. E×RAID exposes per-disk information to an informed file system (namely, I·LFS), providing performance optimizations, more control over redundancy, and improved manageability of storage [12]. Finally, Ganger suggests that a reevaluation of this interface is needed [16], and outlines two relevant case studies: track-aligned extents [35] (which we explore within this paper), and freeblock scheduling [26].

More recent work in the storage community suggests that the next evolution in storage will place disks on a more general-purpose network and not a standard SCSI bus [17]. Some have suggested that these network disks export a higher-level, object-like interface [18], thus moving the responsibilities of low-level storage management from the file system into the drives themselves. Although the specific challenges would likely be different in this context, the fixed object-based interface between file systems and storage will likely provide an interesting avenue for further research into the utility of semantic awareness.

New programming environments: In contrast to integration underneath a traditional file system, other work has focused on incorporating active storage into entirely new parallel programming environments. Recent work on "active disks" includes that by Acharya *et al.* [1], Riedel *et al.* [32], and Amiri *et al.* [2]. Much of this research focuses on how to partition applications across host and disk CPUs to minimize data transferred across system busses.

3 Inferring On-Disk Structures: Fingerprinting the File System

For a semantically smart disk to implement interesting functionality, it must be able to interpret the types of blocks that are being read from and written to disk and specific characteristics of those blocks. For an SDS to be practical, this information must be obtained in a robust manner that does not require human involvement. We consider three alternatives for obtaining this information.

The first approach directly embeds knowledge of the file system within the SDS; thus, the onus of understanding the target file system is placed on the developer of the SDS. The obvious drawbacks are that SDS firmware must be updated whenever the file system is upgraded and the SDS is not robust to changes in the target file system.

With the second approach, the target system informs the SDS of its data structures at run-time; in this case, the responsibilities are placed on the target file system. There are numerous disadvantages with this approach as well. First, and most importantly, the target system must be changed; either the file system (or some other process with access to the same information) must directly communicate with the SDS. Second, a new communication channel outside of existing protocols must be added between the target system and the SDS. Finally, it may be difficult to ensure that the specification communicated to the SDS matches the actual file system implementation.

In the third approach, the SDS automatically infers the file system data structures. The benefits of this approach are many: no specific knowledge about the target file system is required when the SDS is developed; the assumptions made by the SDS about the target file system can be checked when it is deployed; little additional work is required to configure the SDS when it is installed; the SDS can be deployed in new environments with little or no difficulty. We believe that this approach has the most potential for a semantically-smart storage system; thus, we explore how an SDS can automatically acquire layout information with fingerprinting software.

Automatically inferring file system structures bears similarity to several other research efforts in reverse-engineering. For example, researchers have shown that both bit-level machine instruction encodings [22] and the semantic meaning of assembly instructions [10] can be deduced. Others have also developed techniques to identify parameters of the TCP protocol [30], to extract low-level characteristics of disks [34, 38], to determine OS buffer-cache policies [8], and to understand the behavior of a real-time CPU scheduler [31].

3.1 Assumptions

Automatically inferring layout information for an arbitrary file system is a challenging problem. As an important first step, we have developed a utility, called EOF ("Extraction Of Filesystems"), that can extract layout information for FFS-like file systems, either with or without journaling capabilities. We have verified that EOF can identify the data structures employed by Linux ext2 and ext3 as well as NetBSD FFS. Furthermore, EOF should be able to understand a future FFS-like file system that adheres to the following assumptions about the layout of data structures on disk:

General: Disk blocks are statically and exclusively assigned to one of five categories: *data*, *inodes*, *bitmaps for free/allocated data blocks and/or inodes*, *summary information* (*e.g.*, superblock and group descriptors), and *log data.* EOF identifies the block addresses on disk allocated to each category.

Data blocks: A data block may dynamically contain either file data, directory listings, or pointers to other data blocks (*i.e.*, an indirect block). Data blocks are not shared across files. EOF identifies the structure of directory data as well; EOF assumes that each record in a directory data block contains at least the length of the record, the entry name, the length of the entry name, and the inode number for the entry. Each field in a directory entry is assumed to be a multiple of 8 bits. EOF assumes that indirect blocks contain 32-bit pointers.

Inode blocks: An inode block contains N inodes where each inode consumes exactly $1/N$-th of the block. EOF assumes that the definition of each inode field is static over time. EOF identifies the location (or absence) of the following fields within an inode: *size*, *blocks* (the number of data blocks allocated to this inode), *ctime* (the time at which the inode was last changed), *mtime* (the time at which the corresponding data was last changed), *dtime* (the deletion time), *links* (the number of links to this inode), *generation number*, *data pointers* (any number and combination of direct pointers and single, double, and triple indirect pointers) and *dir bits* (bits that change between file and directory inodes). With the exception of *dir bits*, all of the fields that we identify are by default assumed to be a multiple of 32 bits; however, if multiple fields are identified within the same 32 bits (*e.g.*, the *blocks* and *links* fields), then the size of each field is assumed to be the largest multiple of 8 bits that does not lead to overlapping fields.

Bitmap blocks: Bitmaps for data and inodes may either share a single block or be placed in separate blocks. Bits in the (data /inode) bitmap blocks have a one-to-one linear mapping to the data blocks/inodes. The last bitmap block does not have to be entirely valid.

Log data: The log data used by a journaling file system is managed as a circular, contiguous buffer. We make no assumptions about the contents of the log, although we may look into doing so in the future.

The feasibility of inferring on-disk data structures depends upon the assumption that production file systems change slowly over time (if at all). This assumption is likely to hold, given that file system developers have strong motivation to keep on-disk structures the same, so that legacy file systems can continue to operate. Examining file systems of the past and present further corroborates this belief. For example, the on-disk structure of the FFS file system has not changed in nearly 20 years [27]; the Linux ext2 file system has had the same layout since conception; the ext3 journaling file system is backward compatible with ext2 [39]; extensions to FreeBSD FFS are designed so as to avoid on-disk changes [13].

3.2 Algorithm Overview

The EOF software is used as follows. When a new file system is made on an SDS partition, EOF is run on the partition so that the SDS understands the context in which it is being deployed. The basic structure of EOF is that a user-level *probe process* performs operations on the file system, generating controlled traffic streams to disk. The SDS knows each of the high-level operations performed and the disk traffic that should result. By observing which file blocks are written and which bytes within blocks change, the SDS infers which blocks contain each type of file system data structures and which offsets within each block contain each type of field. The SDS can then use this knowledge to configure itself, simultaneously verifying that the target file system behaves as expected.

The SDS must be able to correlate the traffic it observes with the file system operations performed by the probe process. This correlation requires two pieces of functionality. First, the probe process must ensure that all blocks from an operation have been flushed out of the file system cache and written to the SDS. To ensure this, the probe process unmounts the file system; however, unmounting (and re-mounting) is used sparingly since it increases the running time of EOF. Second, the probe process must occasionally inform the SDS that a specific operation has ended. The probe process communicates to the SDS by writing a distinct pattern to a *fencepost* file; the SDS looks for this known pattern in the resulting traffic to find the message from the probe process.

Two general techniques are used within EOF to identify blocks and inode fields. First, to identify data blocks, the SDS always looks for a known pattern that the probe process writes in test files. Second, to classify all other blocks and fields, the SDS attempts to isolate a unique, unclassified block that is written by one operation, across a set of operations, or by some operations but not by others.

3.3 Algorithm Phases

EOF is composed of five phases. First, EOF isolates the summary blocks and the log file. Next, EOF identifies data blocks and data bitmaps. Then, EOF looks for inodes and inode bitmaps. After all blocks have been classified, EOF isolates the inode fields. Finally, EOF identifies the fields within directory entries.

3.3.1 Bootstrapping (Phase 0)

The goal of bootstrapping is to isolate the blocks that are frequently written in later phases so that they can be filtered from the blocks of interest. Thus, phase 0 isolates summary blocks, the log file, and inode and data blocks for the fencepost file, the test directory, and a few test files.

First, the probe process creates the fencepost file and a number of test files within a test directory; the SDS identifies the data blocks associated with each file by searching for the known patterns. Second, EOF identifies the blocks belonging to the log file, if it exists. In this step, the probe process synchronously appends data with a known pattern to one of the test files. The SDS observes many blocks of meta-data; those blocks that are written to in a circular pattern belong to the log (if no block traffic matches this pattern, then EOF infers that the file system does not perform journaling). Third, EOF identifies the summary blocks; the probe process unmounts the file system and the written blocks that have not been classified as log data are identified as summary blocks.

To isolate the inode blocks that are repeatedly written, the probe process performs a `chmod` on the fence-post file, the test directory, and the test files; in each case, only the inode of each is written, allowing it to be classified. The data blocks belonging to the test directory are identified by changing the name of each test file; these blocks are the only previously unidentified blocks written. Finally, to determine if separate bitmap blocks are used for data and inode blocks (*i.e.*, as in Linux ext2 and ext3) or if a single bitmap block is shared between both (*i.e.*, as in NetBSD FFS), EOF creates a new file; whether the SDS observes one or two unclassified blocks allows it to determine whether bitmap blocks are shared or kept separate for data and inodes. To simplify our presentation, in the remainder of our discussion we consider only the case where data and inode bitmaps are in separate blocks; however, EOF correctly handles the shared case, in which case, EOF also isolates the specific bits in the shared bitmap block devoted to inode or data block state.

3.3.2 Data and Data-bitmap blocks (Phase 1)

EOF continues by identifying all the blocks on disk containing either data or data bitmaps. To isolate these blocks, the probe process appends a few blocks of data with a known pattern to each of the test files. All blocks that do not match the known pattern and are not yet classified are assumed to be either data-bitmap blocks or indirect-pointer blocks. EOF differentiates between the two by inferring that blocks written by two different files must be data-bitmap blocks. Care is taken to create small enough files such that no single file fills a bitmap block; the last bitmap block is a special case since a smaller than expected file can completely fill it. To cleanup from this phase, the test files are deleted.

3.3.3 Inodes and Inode-bitmap blocks (Phase 2)

Identifying the inodes and their bitmaps requires creating many new files. Two distinct steps are required. First, the probe process creates many new files, which causes both the inodes and inode bitmaps to be modified. Second, the probe process performs a `chmod` on the files, which causes the inodes but not the inode bitmaps to be written. Thus, the inodes and inode bitmaps can be distinguished from each other. This phase also calculates the size of each inode; this is performed by recording the number of times each block is identified as an inode and dividing the block size by the observed number of inodes in a block.

3.3.4 Inode Fields (Phase 3)

At this point, EOF has classified all blocks on disk as belonging to one of the five categories of data structures. The next phase identifies fields within inodes by observing those fields that do or do not change across operations. For brevity, we do not describe how EOF infers the blocks, links, and generation number fields.

The first inode fields that EOF identifies are the file size and times; this requires five steps. First, the probe process creates a file; the SDS stores the inode data to compare it to the inode data written in the next steps. Second, the probe process overwrites the file data; the only inode fields that change are those related to time. Third, the probe process appends a small amount of data to the file such that a new data pointer is not added; at this point, the size field can be identified as the only data that changed in step 3 but not step 2. Fourth, the probe process performs an operation to change the inode without changing the file data (*e.g.*, adding a link or changing the permissions); this allows the SDS to isolate *mtime* (which is not changed in this step) from *ctime* (which is changed). Finally, the file is deleted so that the deletion-time field is observed.

EOF next identifies the location and the level of the data pointers in the inode. The probe process repeatedly appends to a file while the SDS observes which bytes in the inode change (other than those that changed in the previous step). EOF infers the location of indirect pointers (and so forth) by observing when an additional "data" block is written and no additional pointer is updated in the inode. To improve performance, rather than write every block, the probe process seeks a progressively larger amount: the seek distance starts at one block and increases by the size handled by the currently detected indirection level.

Finally, EOF isolates the inode bit fields that designate directories. The probe process alternately creates files and directories. The SDS keeps two histograms: one for file and one for directory inodes; in the histogram, EOF records the count of times each bit in the inode type was zero. To determine the directory fields, EOF isolates all bits that were always 0 for files and always 1 for directories (and vice versa). These bits and their corresponding values are then considered to identify files versus directories. Soft link bits are identified in a similar manner.

3.3.5 Directory Entries (Phase 4)

In its final phase, EOF identifies the structure of entries within a directory. First, EOF infers the offsets of the entry name and the name length. To do this, the probe process creates a file with a known name; the SDS searches for this name in the directory data block as well as the field designating the length of this name. For validation, this file is deleted and the step is repeated numerous times for filenames of different lengths. Second, EOF finds the location of the record length, using the assumption that the length of the last record contains the remaining space for the directory data block and that this length is reduced when a new record is added. Thus, the probe process creates additional files and the SDS simply records the offsets that change in the previous entries. Finally, the offset of the inode number is found using the assumption that each directory contains an entry for itself (*i.e.*, "."). In this step, the probe process creates two empty directories; the SDS isolates the inode offset by recording the differences across the data blocks of those two directories.

3.4 Assertion of Assumptions

The major challenge with automatic inferencing is to ensure that the SDS has correctly identified the behavior of the target file system. To be robust to a new file system not meeting these assumptions, EOF has mechanisms to detect when an assumption fails; in this case, the file system is identified as non-supported and the SDS operates correctly, but without using semantic knowledge. For example, if more blocks than expected are written in a specific step, or if specific blocks are not observed, EOF detects this as a violation. We have verified that violations are identified appropriately when EOF is run upon non-FFS file systems (*e.g.*, msdos, vfat, and reiserfs)."

An additional benefit of using EOF to configure an SDS is that file system bugs may be identified. For example, running EOF on ext3 in Linux 2.4 isolated two bugs. First, the SDS observed incomplete traffic in key steps; this problem was tracked back to an ext3 bug in which data written within 30 seconds prior to an unmount is not always flushed to disk [28]. Second, the probe process noted an error when all of the inodes were allocated; in this case, ext3 incorrectly marks the file system as dirty [25]. Thus, EOF enables checks of the file system that are not easily obtained with other methods.

4 Exploiting Structural Knowledge: Classification and Association

The key advantage of an SDS is its ability to identify and utilize important properties of each block on the disk. These properties can be determined through direct and indirect classification as well as through association. With *direct classification*, blocks are easily identified by their location on disk. With *indirect classification*, blocks are identified only with additional information; for example, to identify directory data or indirect blocks, the corresponding inode must also be examined. Finally, with *association*, a data block and its inode are connected.

In many cases, an SDS also requires functionality to identify when a change has occurred within a block. This functionality is implemented via block differencing. For example, to infer that a data block has been allocated, a single-bit change in the data bitmap must be observed. Change detection is potentially one of the most costly operations within an SDS for two reasons. First, to compare the current block with the last version of the block, the SDS may need to fetch the old version of the block from disk; however, to avoid this overhead, a cache of blocks can be employed. Second, the comparison itself may be expensive: to find the location of a difference, each byte in the new block must be compared with the corresponding byte in the old block. We quantify these costs in Section 6.

4.1 Direct Classification

Direct classification is the simplest and most efficient form of on-line block identification for an SDS. The SDS determines the type of the block by performing a simple bounds check to calculate into which set of block ranges a particular block falls. In an FFS-like file system, the superblock, bitmaps, inodes, and data blocks are identified using this technique.

4.2 Indirect Classification

Indirect classification is required when the type of a block can vary dynamically and thus simple direct classification cannot precisely determine the type of block. For example, in FFS-like file systems, indirect classification is used to determine whether a data block is file data, directory data, or some form of indirect pointer block (*e.g.*, a single, double, or triple indirect block). To illustrate these concepts we focus on how directory data is differentiated from file data; the steps for identifying indirect blocks versus pure data are similar.

Identifying directory data: The basic challenge in identifying whether a data block belongs to a file or a directory is to track down the inode that points to this data and check whether its type is a file or a directory. To perform this tracking, the SDS *snoops* on all inode traffic to and from the disk: when a directory inode is observed, the corresponding data block numbers are inserted into a hash table. The SDS removes data blocks from the hash table by observing when those blocks are freed (*e.g.*, by using block differencing on the bitmaps). When the SDS must later identify a block as a file or directory block, its presence in this table indicates that it is directory data. We now discuss two complications with this approach.

First, the SDS cannot always guarantee that it can correctly identify blocks as files or directories. Specifically, when a data block is not present in the hash table, the SDS infers that the data corresponds to a file; however, in some cases, the directory inode may not have yet been seen by the SDS and as a result is not yet in the hash table. Such a situation may occur when a new directory is created or when new blocks are allocated to existing directories; if the file system does not guarantee that inode blocks are written before data blocks, the SDS may incorrectly classify newly written data blocks. This problem does not occur when classifying data blocks that are read. In this case, the file system must read the corresponding inode block before the data block (to find the data block number); thus, the SDS will see the inode first and correctly identify subsequent data blocks.

Whether or not transient misclassification is a problem depends upon the functionality provided in the SDS. For instance, if an SDS simply caches directory blocks for performance, it can likely tolerate a temporary inaccuracy. However, if the SDS requires accurate information for correctness, there are two ways it can be ensured. The first option is to guarantee that the file system above writes inode blocks before data blocks; this is true by default in FFS (before soft updates [36]) and in Linux ext2 when mounted in synchronous mode. The second option is to buffer writes until the time when the classification can be made; this *deferred classification* occurs when the corresponding inode is written to disk or when the data block is freed, as can be inferred by monitoring data bitmap traffic.

Second, the SDS may perform excess work if it obliviously inserts all data blocks into the hash table whenever a directory inode is read and written since this inode may have recently passed through the SDS, already causing the hash table to be updated. Therefore, to optimize performance, the SDS can infer whether or not a block has been added (or modified or deleted) since the last time this directory inode was observed, and thus ensure that only those blocks are added to (or deleted from) the hash table. This process of *operation inferencing* is described in detail in Section 5.

Identifying indirect blocks: The process for identifying indirect blocks is almost identical to that for identifying directory data blocks. In this case, the SDS tracks new indirect block pointers in all inodes being read and written. By maintaining a hash table of all single, double, and triple indirect block addresses, an SDS can determine if a data block is an indirect block.

4.3 Association

The most useful association is to connect data blocks with their inodes; for example, this allows the size or creation date of a file to be known by the SDS. Association can be achieved with a simple but space-consuming approach. Similar to indirect classification, the SDS snoops on all inode traffic and inserts the data pointers into an address-to-inode hash table. One concern about such a table is size; for accurate association, the table grows in proportion to the number of unique data blocks that have been read or written to the storage system since the system booted. However, if approximate information is tolerated by the SDS, the size of this table can be bounded.

5 Detecting High-Level Behavior: Operation Inferencing

Block classification and association provide the SDS with an efficient way for identifying special kinds of blocks; however, operation inferencing is necessary to understand the semantic meaning of the changes observed in those blocks. We now outline how an SDS can identify file system operations by observing certain key changes.

One challenge with operation inferencing is that the SDS must distinguish between blocks which have a valid "old version" and those that do not. For instance, when a newly allocated directory block is written, it should not be compared to the old contents of the block since the block contained arbitrary data. To identify when to use the old versions, the SDS uses a simple insight: when a meta-data block is written without being read, the old contents of the block are not relevant. To detect this situation, the SDS maintains a hash table of meta-data block addresses that have been read sometime in the past. Whenever a meta-data block is read, it is added to this list; whenever the block is freed (as indicated by a block bitmap reset), it is removed from the list. For example, when a block allocated to a data file is freed and reallocated to a directory, the block address will not be present in the hash table, and hence the SDS will not use the old contents.

For illustrative purposes, in this section we examine how the SDS can infer file create and delete operations. The discussion below is specific to ext2, although similar techniques can be applied to other FFS-like file systems.

5.1 File Creates and Deletes

There are two steps in identifying file creates and deletes. The first is the actual detection of a create or delete; the second is determining the inode that has been affected. We describe three different detection mechanisms and the corresponding logic for determining the associated inode.

The first detection mechanism involves the inode block itself. Whenever an inode block is written, the SDS examines it to determine if an inode has been created or deleted. A valid inode has a non-zero modification time and a zero deletion time. Therefore, whenever the modification time changes from zero to non-zero or the deletion time changes from non-zero to zero, it means the corresponding inode was newly made valid, *i.e.*, created. Similarly, a reverse change indicates a newly freed inode, *i.e.*, a deleted file. A second indication is a change in the version number of a valid inode, which indicates that a delete followed by a create occurred. In both cases, the inode number is calculated using the physical position of the inode on disk (on-disk inodes do not contain inode numbers).

The second detection mechanism involves the inode bitmap block. Whenever a new bit is set in the inode bitmap, it indicates that a new file has been created corresponding to the inode number represented by the bit position. Similarly, a newly reset bit indicates a deleted file.

The update of a directory block is a third indication of a newly created or deleted file. When a directory data block is written, the SDS examines the block for changes from the previous version. If a new directory entry (`dentry`) has been added, the name and inode number of the new file can be obtained from the `dentry`; in the case of a removed `dentry`, the old contents of the `dentry` contain the name and inode number of the deleted file.

Given that any of these three changes indicate a newly created or deleted file, the choice of the appropriate mechanism (or combinations thereof) depends on the functionality being implemented in the SDS. For example, if the SDS must identify the deletion of a file, immediately followed by the creation of another file with the same inode number, the inode bitmap mechanism does not help, since the SDS may not observe a change in the bitmap if the two operations are grouped due to a delayed write in the file system. In such a case, using modification times and version numbers is more appropriate. Similarly, if the name of the newly created or deleted file must be known, the directory block-based solution is the most efficient.

5.2 Other File System Operations

The general technique of inferring logical operations by observing changes to blocks from their old versions can help detect other file system operations as well. We note that in some cases, for a conclusive inference on a specific logical operation, the SDS must observe correlated changes in multiple meta-data blocks. For example, the semantically-smart disk system can infer that a file has been renamed when it observes a change to a directory block entry such that the name changes but the inode number stays the same; note that the version number within the inode must stay the same as well. Similarly, to distinguish between the creation of a hard link and a normal file, both the directory entry and the file's inode must be examined.

6 Evaluation

In this section, we answer three important questions about our SDS framework. First, what is the cost of fingerprinting the file system? Second, what are the time overheads associated with classification, association, and operation inferencing? Third, what are the space overheads? Before proceeding with the evaluation, we first describe our experimental environment.

6.1 Platform

To prototype an SDS, we employ a software-based infrastructure. Our implementation inserts a pseudo-device driver into the kernel, which is able to interpose on traffic between the file system and the disk. Similar to a software RAID, our prototype appears to file systems above as a device upon which a file system can be mounted.

The primary advantage of our prototype is that it observes the same information and traffic stream as an actual SDS, with no changes to the file system above. However, our current infrastructure differs in three important ways from a true SDS. First, and most importantly, our prototype does not have direct access to low-level drive internals; using such information is thus made more difficult. Second, because the SDS runs on the same system as the host OS, there may be interference due to competition for resources; in our initial case studies, we do not believe this to be of prime importance. Third, the performance characteristics of the microprocessor and memory system may be different than an actual SDS; however, high-end storage arrays already have significant processing power, and this processing capacity will likely trickle down into lower-end storage systems.

We have experimented with our prototype SDS in the Linux 2.2, Linux 2.4, and NetBSD 1.5 operating systems, underneath of the ext2, ext3, and FFS file systems, respectively. Most experiments in this paper are performed on a processor that is "slow" by modern standards, a 550 MHz Pentium III processor, with either an 10K-RPM IBM 9LZX or 10K-RPM Quantum Atlas III disk. In some experiments, we employ a "fast" system, comprised of a 2.6 GHz Pentium IV and a 15K-RPM Seagate Cheetah disk, to gauge the effects of technology trends.

6.2 Off-line: Layout Discovery

In this subsection, we show that the time to run the fingerprinting tool, EOF, is reasonable for modern disks. Given that EOF only needs to run once for each new file system, the runtime of EOF does not determine the common case performance of an SDS; however, we do not want the runtime of EOF to be prohibitive, especially as disks become larger. One potential solution is parallelism: we believe

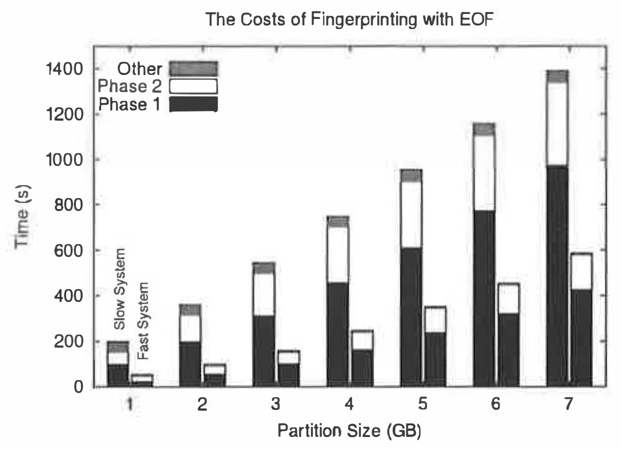


Figure 1: **The Costs of Fingerprinting.** *The figure presents the time breakdown of the fingerprinting on both the "slow" system (IBM disk), and the "fast" system, both running underneath Linux ext2. Along the x-axis, we vary the size of the partition that is fingerprinted, and the y-axis shows the time taken per phase.*

that the time-consuming components of EOF are parallelizable, which would reduce run-time on disk arrays.

Figure 1 presents a graph of the time to run EOF on a single-disk partition as the size of the partition is increased. We show performance results for both the "slow" system with the IBM disk and the "fast" system. The graph shows that Phase 1, which determines the locations of data blocks and data bitmaps, and Phase 2, which determines the locations of inode blocks and inode bitmaps, dominate the total cost of fingerprinting. The time for these two phases increases linearly with the size of the partition, requiring approximately 190 seconds per GB on the slow system, and 81 seconds per GB on the fast system. Comparing performance across the two systems, we conclude that increases in sequential disk performance directly improve EOF fingerprinting time. The other phases require a small amount of time regardless of partition size.

6.3 On-line: Time Overheads

Classification, association, and operation inferencing are potentially costly operations for an SDS. In this subsection, we employ a series of microbenchmarks to illustrate the various costs of these actions. The results of our experiments on an SDS underneath of Linux ext2 are presented in Table 1. For each action and microbenchmark we consider two cases. In the first case, the file system is mounted synchronously, ensuring that meta-data operations reach the SDS in order and thus allowing the SDS to guarantee correct classification with no additional effort; synchronous mounting in Linux ext2 is quite similar to traditional FFS in its handling of meta-data updates. In the second case, the file system is mounted asynchronously; in this case, to guarantee correct classification and association the SDS must perform operation inferencing. The microbenchmarks perform basic file system operations, including file and directory creates and deletes, and we report the per-file or per-directory overhead of the action that is under test.

From our experiments, we make a number of observations. First, most operations tend to cost on the order of tens of microseconds per file or directory. Although some of the operations do require nearly 300 μs to complete, most of this cost is due to a per-block cost; for example, operation inferencing in synchronous mode with the Create$_{32}$ workload takes roughly 280 μs, which corresponds to a 34 μs base cost (as seen in the Create$_0$ workload) plus a cost of approximately 30 μs for each 4 KB block. Thus, although the costs rise as file size increases, the SDS incurs only a small per-block overhead compared to the actual disk writes, each of which may take some number of milliseconds to complete. Second, in most cases, the overheads when the ext2 file system is run in asynchronous mode are much lower than when run in synchronous mode. In asynchronous mode, numerous updates to meta-data blocks are batched and thus the costs of block differencing are amortized; in synchronous mode, each meta-data operation is reflected through to the disk system, incurring much higher overhead in the SDS. Third, we observe that in synchronous mode, classification is less expensive than association which is less expensive than inferencing; an SDS should take care to employ only those actions that are needed to implement the desired functionality.

6.4 On-line: Space Overheads

An SDS may require additional memory to perform classification, association, and operation inferencing; specifically, hash tables are required to track mappings between data blocks and inodes whereas caches are needed to implement efficient block differencing. We now quantify these memory overheads under a variety of workloads.

Table 2 presents the number of bytes used by each hash table to support classification, association, and operation inferencing. The sizes are the maximum reached during the run of a particular workload: NetNews [37], PostMark [24], and the modified Andrew benchmark [29]. For NetNews and PostMark, we vary workload size, as described in the caption.

From the table, we see that the dominant memory overhead occurs in an SDS performing block-inode association. Whereas classification and operation inferencing require table sizes that are proportional to the number of unique meta-data blocks that pass through the SDS, association requires information on every unique data block that passes through. In the worst case, an entry is required

	Indirect Classification		Block-Inode Association		Operation Inferencing	
	Sync	Async	Sync	Async	Sync	Async
$Create_0$	1.7	3.2	1.9	3.3	33.9	3.2
$Create_{32}$	60.6	3.8	324.4	16.4	279.7	3.8
$Delete_0$	4.3	3.6	6.7	3.9	50.9	3.6
$Delete_{32}$	37.8	6.9	80.1	28.8	91.0	6.9
Mkdir	56.3	8.6	63.6	11.1	231.9	8.6
Rmdir	49.9	106.2	57.8	108.5	289.4	106.2

Table 1: **SDS Time Overheads.** *The table breaks down the costs of indirect classification, block-inode association, and operation inferencing. Different microbenchmarks (one per row) stress various aspects of each action. The Create benchmark creates 1000 files, of size 0 or 32 KB, and the Delete benchmark similarly deletes 1000 such files. The Mkdir and Rmdir benchmarks create or remove 1000 directories, respectively. Each result presents the average overhead per operation in μs (i.e., how much extra time the SDS takes to perform classification, association, or inferencing). The experiments were run upon the "slow" system with the IBM 9LZX disk, with Linux ext2 mounted synchronously* (**Sync**) *or asynchronously* (**Async**).

	Indirect Classification	Block-Inode Association	Operation Inferencing
$NetNews_{50}$	68.9 KB	1.19 MB	73.3 KB
$NetNews_{100}$	84.4 KB	1.59 MB	92.3 KB
$NetNews_{150}$	93.3 KB	1.91 MB	105.3 KB
$PostMark_{20}$	3.45 KB	452.6 KB	12.6 KB
$PostMark_{30}$	3.45 KB	660.7 KB	16.2 KB
$PostMark_{40}$	3.45 KB	936.4 KB	19.9 KB
Andrew	360 B	3.54 KB	1.34 KB

Table 2: **SDS Space Overheads.** *The table presents the space overheads of the structures used in performing classification, association, and operation inferencing, under three different workloads (NetNews, PostMark, and the modified Andrew benchmark). Two of the workloads (NetNews and PostMark) were run with different amounts of input, which correspond roughly to the number of "transactions" each generates (i.e., $NetNews_{50}$ implies 50,000 transactions were run). Each number in the table represents the maximum number of bytes stored in the requisite hash table during the benchmark run (each hash entry is 12 bytes in size). The experiment was run on the "slow" system with Linux ext2 in asynchronous mode on the IBM 9LZX disk.*

for every data block on the disk, corresponding to 1 MB of memory for every 1 GB of disk space. Although the space costs of tracking association information are high, we believe they are not prohibitive. Further, if memory resources are scarce, the SDS can choose to either tolerate imperfect information (if possible), or swap portions of the table to disk.

In addition to the hash tables needed to perform classification, association, and operation inferencing, a cache of "old" data blocks is useful to perform block differencing effectively; recall that differencing is used to observe whether pointers have been allocated or freed from an inode or indirect block, to check whether time fields within an inode have changed, to detect bitwise changes in a bitmap, and to monitor directory data for file creations and deletions. The performance of the system is sensitive to the size of this cache; if the cache is too small, each difference calculation must first fetch the old version of the block from disk. To avoid the extra I/O, the size of the cache must be roughly proportional to the active meta-data working set. For example, for the $PostMark_{20}$ workload, we found that the SDS cache should contain approximately 650 4 KB blocks to hold the working set. When the cache is smaller, block differencing operations often go to disk to retrieve the older copy of the block, increasing run-time for the benchmark by roughly 20%.

7 Case Studies

In this section, we describe our case studies, each implementing new functionality in an SDS that would not be possible to implement within a drive or RAID without semantic knowledge. Some of these case studies could be built into the file system proper; however, implementing "file-system like" functionality in the storage system is one the many advantages of semantic intelligence, as it allows storage-system manufacturers to augment their products with a much broader range of capabilities.

Due to space limitations, we cannot fully describe each of the case studies in this paper; instead, we highlight the functionality each case study implements, present a brief performance evaluation, and conclude by analyzing the complexity of implementing said functionality within an SDS. Each performance evaluation is included to demonstrate that interesting functionality can be implemented effectively within an SDS; we leave more detailed performance studies as future work. One theme we explore within this section is the usage of "approximate" information, *i.e.*, scenarios in which an SDS can be wrong in its understanding of the file system.

7.1 The Case Studies

Track-Aligned Extents: As proposed by Schindler *et al.* [35], track-aligned extents (traxtents) can improve disk access times by placing medium-sized files within tracks and thus avoiding track-switch costs. Given the detailed level of knowledge that a traxtents-enabled file system requires of the underlying disk (*i.e.*, the mapping of logical block numbers to physical tracks), traxtents are a natural candidate for implementation within an SDS, where this information is readily obtained.

The fundamental challenge of implementing traxtents in an SDS instead of the file system is in adapting to the *policies* of the file system outside of the file system; specifically, a Traxtent SDS must influence file system allocation and prefetching, *e.g.*, mid-sized files must be

	Without Prefetching	With Prefetching
ext2	10.3 MB/s	10.2 MB/s
+Traxtent SDS	12.2 MB/s	14.2 MB/s

Table 3: **Track-Aligned Extents.** *The table shows the bandwidth obtained when reading 100 files in a randomized order. Each file is roughly the size of a track, in this case 328 KB. We examine both default and track-aligned allocation, varying whether track-sized prefetching is enabled within the SDS. This experiment was run upon the "slow" system running Linux 2.2 with the ext2 file system mounted asynchronously upon the Quantum Atlas III disk.*

	TPC-B_{20}	TPC-B_{100}
FFS	25.04	45.27
+LRU SDS	26.52	48.58
+File-Aware Caching SDS	3.88	20.58

Table 4: **File-Aware Caching.** *The table shows the time in seconds it takes to execute 20,000 and 100,000 TPC-B transactions. In all experiments, transactions first run to warm up the system; then a large scan is run, followed by another series of transactions, which are timed. The table compares NetBSD FFS on a standard disk, on an SDS with a 100 MB LRU-managed cache, and on an SDS with a 100 MB file-aware cache. All experiments are run on the "slow" system and the IBM 9LZX disk.*

allocated such that consecutive data blocks do not span track boundaries and accesses must be in track-sized units.

There are three components of interest within the Traxtent SDS implementation. First, when the bitmap blocks are first read by the file system, the SDS marks the bitmap corresponding to the last block in each track as allocated, (a similar technique is used by Schindler *et al.*). Although this wastes a small portion of the disk, this "fake" allocation influences the file system to allocate files such that they do not span tracks. Second, if the file system still decides to allocate a file across tracks, the SDS dynamically remaps those blocks to a track-aligned locale, similar to the block remapping of Loge and other smart disks [15, 40]. One major difference is that the SDS only remaps blocks that are a part of mid-sized files that benefit from track-alignment, whereas non-semantically aware disks cannot make such a distinction. Third, the Traxtent SDS performs additional prefetching to ensure accesses are not smaller than a track. Linux ext2 (and FFS as well) prefetches very few blocks when a file is initially read; therefore, when the Traxtent SDS observes a read to the first block of a track-aligned file, it requests the remainder of the track and places the data blocks in its cache.

The Traxtent SDS relies upon one piece of exact information for correctness: the location of bitmap blocks, which it marks to "trick" the file system into track-aligned allocation. However, given that this information is static, it can be obtained reliably with EOF and with little performance cost at runtime. The indirect classification of file data as belonging to medium-sized files can be occasionally incorrect, since their remapping is only for performance and not correctness. Table 3 shows that the Traxtent SDS with prefetching results in roughly a 40% improvement in bandwidth for medium-sized files.

Structural Caching: We next discuss the use of semantic information in caching within an SDS. Simple LRU management of a disk cache is likely to duplicate the contents of the file system cache [41, 43], and thereby wastes memory in the storage system. This waste is particularly onerous in storage arrays, due to their large amounts of memory. In contrast, an SDS can use its structural understanding of the file system to cache blocks more intelligently, and thus avoid wasteful replication. We explore the caching of blocks in both volatile memory (DRAM) and non-volatile memory (NVRAM), as each presents unique opportunities for optimization.

We first examine a simple optimization that avoids worst-case LRU behavior. This File-Aware Caching SDS (FAC SDS) exploits knowledge of file size to selectively cache blocks from files that are small enough to fit into the available cache, or that are from files that are not being accessed sequentially. This strategy avoids caching blocks from large files that are being scanned and would otherwise flush the cache of all other blocks.

To implement file-aware caching, the FAC SDS identifies cacheable blocks using indirect classification and association; in this case, the hash table holds block addresses that correspond to those files that meet the caching criteria. As described previously, this may cause the SDS to misclassify blocks in those cases when the file inode is written to disk after the data blocks. The FAC SDS also keeps a small amount of state per active file in order to detect sequential access patterns.

Table 4 shows the performance of the FAC SDS under a database workload. In this scenario, we run TPC-B transactions, and periodically intersperse large file scans into the system, thus emulating a system running mixed interactive and batch transactions. Whereas the large scan flushes the contents of a traditional LRU-managed cache (and hence degrades performance for subsequent transactions), the file-aware cache does not cache blocks from large scans, thus keeping the transactional tables in SDS memory and improving performance.

We next examine how an SDS can use semantic knowledge to store important structures in non-volatile memory. We explore two different possibilities. In the first, we exploit semantic knowledge to store the ext3 journal in NVRAM. To implement the Journal Caching SDS (JC SDS), the SDS must recognize traffic to the journal and redirect it to the NVRAM. Doing so is straightforward, as the EOF tool determines which blocks belong to the journal. Thus, by classifying and then caching data reads

	Create	Create+Sync
ext3	4.64	32.07
+LRU_8 SDS	5.91	11.96
+LRU_{100} SDS	2.39	3.35
+Journal Caching SDS	4.66	6.35

Table 5: **Journal Caching.** *The table shows the time to create 2000 32-KB files, under ext3 without an SDS, with an SDS that performs LRU NVRAM cache management using either 8 MB or 100 MB of cache, and with the Journal Caching SDS storing an 8 MB journal in NVRAM. The* **Create** *benchmark performs a single* `sync` *after all of the files have been created, whereas the* **Create+Sync** *benchmark performs a* `sync` *after each file creation, thus inducing a journaling-intensive workload. These experiments are run on the "slow" system running Linux 2.4 and utilizing IBM 9LZX disk.*

	Create	Read	Delete	PostMark
FFS	73.61	5.14	64.41	230.0
+LRU_8 SDS	1.67	211.10	1.32	333.0
+LRU_{100} SDS	1.67	3.51	4.32	12.0
+MDC SDS	1.76	11.34	0.91	19.0

Table 6: **Meta-data Caching.** *The left three columns of the table show the time in seconds to complete each phase of the LFS microbenchmark [33] (in this experiment, the LFS benchmark creates, reads, and deletes 6500 1-KB files). The right column shows the total time in seconds for the PostMark benchmark, run with 5000 files, 5000 transactions, and 71 directories. The rows compare performance under NetBSD FFS on the "slow" system and IBM disk without an SDS, with an SDS that performs LRU NVRAM cache management using either 8 MB or 100 MB of cache, and with the MDC SDS strategy.*

and writes to the journal file, the SDS can implement the desired functionality.

In the second, we place all of the meta-data (bitmaps, inodes, indirect blocks, and directories) of NetBSD FFS in NVRAM. Inodes and bitmaps are identified by their location on the disk. Pointer blocks and directory data blocks are identified with indirect classification, which can occasionally miss blocks. Here again we exploit the fact that approximate information is adequate; the SDS writes unclassified blocks to disk and not NVRAM, until it observes the corresponding inode. To track meta-data blocks, the Meta-data Caching SDS (MDC SDS) uses an additional map to record their in-core location.

Tables 5 and 6 show the performance of the JC SDS and the MDC SDS. In both cases, simple NVRAM caching of structures such as a journal or file system meta-data are effective at reducing run times, sometimes dramatically, by greatly reducing the time taken to write blocks to stable storage. An LRU-managed cache can also be effective in this case, but only when the cache is large enough to contain the working set. One of the main benefits of structural caching in NVRAM is that the size of the cached structures is known to the SDS and thus guarantees effective cache utilization. A hybrid may combine the best of both worlds, by storing important structures such as a journal or other meta-data in NVRAM, and managing the rest of available cache space in an LRU fashion.

In the future, we plan to investigate other ways in which semantic information can be used to improve storage-system cache management. For example, an SDS can use certain types of meta-data updates (such as last-accessed-time updates in an inode) in order to ascertain what files are likely to be in the file system cache above. Prefetching within an SDS is also likely to be more intelligent, as the system has file awareness and thus can make a better guess as to which block will next be read. Finally, blocks that have been deleted can be removed from the cache, thus freeing space for other live blocks.

Secure Deletion: With advanced magnetic force scanning tunneling microscopy (STM), a person with physical access to a drive (and a lot of time) can potentially extract sensitive data that the user had "deleted" [20]. In this case study, we explore a "secure-deleting" SDS, that is, a disk that guarantees that file data from deleted files is truly unrecoverable. Previous approaches have (incorrectly) placed such functionality within the file system by over-writing deleted file blocks multiple times with various patterns [6]. However, this does not guarantee that the data is removed from the disk; other copies of various data blocks may exist, due to bad-block remapping or other storage system optimizations [20, 42]. Further, multiple consecutive file-system writes may not reach the disk media due to NVRAM buffering [5]. An SDS is the only locale where a secure delete can be implemented, since it can ensure that no stray copies of data exist and that over-writes are performed on the disk.

Because of the nature of this case study, approximate or incorrect information about which blocks have been deleted is not acceptable. The Secure-deleting SDS recognizes deleted blocks through operation inferencing and then overwrites those blocks with different data patterns a specified number of times. Since the file system may reallocate these blocks to a different file and possibly write the block with fresh contents in the meantime, the SDS tracks deleted blocks and queues writes to those blocks until the overwrite has finished. Also note that we currently must mount the ext2 file system synchronously for secure deletion to operate correctly; we are investigating techniques to relax this requirement as a part of future work.

Table 7 shows the overhead incurred by an SDS, as a function of the number of over-writes; the more over-writes performed, the less likely the data will be recoverable. Although a noticeable price is paid for the secure-delete functionality, this loss may be acceptable to highly-sensitive applications requiring such security. Performance could be further improved by delaying the secure-

Once the disk has checked and executed the requested operation op, it sends back a response $resp$ together with $h(resp, s)$. Here $resp$ contains data (if the request was a read) or simply an acknowledgment (if the request was a write). The client verifies that $h(resp, s)$ was computed correctly, which prevents responses from being forged.

For simplicity, we presented this example without encryption for privacy. One simple way of adding encryption involves the server also giving the client a session key e and a token, which is e encrypted under k; the client and disk encrypt their messages using e, prepending the token in the clear so the disk can figure out which session key to use.

2.2 Revoking capabilities

A revocation is required whenever a client should no longer have the access granted by a previously issued capability—due, for example, to a change in file permissions, or a file truncation or deletion. We seek a revocation scheme that is memory efficient, so that it can ideally be implemented in existing network-attached disks by simply changing their firmware (rather than, say, adding more hardware to them). The difficulty is that such disks have little memory and most of it is used to cache data (a typical cache size is 4 MB). It is thus important that the adopted scheme not take much memory: tens of kilobytes would be excellent, while megabytes would probably be too much.

Earlier schemes described in the literature, such as NASD [10] and SCARED [19], use object version numbers for revocations. In these systems, capabilities confer access only to a particular object version, so incrementing an object's version number suffices to revoke all old capabilities for it. Although this makes sense for variable-length objects, whose headers must be read first to find out where the desired data actually resides on disk, it is problematic for blocks: changing the permissions of a file would require updating a potentially large number of version numbers. For example, a 512 MB file could require updating 1 million version numbers, which would span 8 thousand blocks assuming 32 bits per version number.

Thus, we need a scheme that allows direct revocations of capabilities, without having to access the blocks to which the capability refers. A simple and economical method is to assign capability ID's to each capability so that the disk can keep track of valid capabilities through a bit vector. By doing so, it is possible to keep track of 524,288 capabilities with 64 KB, a modest amount of memory.

As a further optimization, we can assign the same ID to different capabilities, thereby reducing the number of possible revocations that need to be kept track of, but at the cost of not being able to independently revoke capabilities that share an ID. It makes sense to group together all the capabilities describing the same kind of access to different parts of one file, because they are almost always revoked together (the exception being partial truncation). One can further group together capabilities for the same file with different access modes; this makes file creation and deletion cheaper at the cost of making permission changes (*e.g.*, `chmod`), which are rare, slightly more expensive.

This straightforward approach has a problem though: once an ID is revoked, its bit in the revocation vector must be kept set *forever*, lest some attacker hold the revoked capability and much later illegally use it again if the bit were cleared. Therefore, no matter how large the number of ID's, the system will sooner or later run out of them.

One way to solve this difficulty is to change k (the secret key shared by the disk and metadata server) whenever ID's run out, causing all existing capabilities to become stale. The disk's bit vector can now be cleared without fear of old invalidated capabilities springing back to life and posing a security threat.

This scheme has the unfortunately problem that when k is changed all capabilities become stale at once and will be rejected by the disk, so all clients need to go back to the metadata server to get fresh capabilities. Therefore, the metadata server may get overloaded with a shower of requests in a short period. We call this the *burstiness problem.*

We solve the burstiness problem by using *capability groups.* The basic idea is to place capabilities into groups, and to invalidate groups when the system needs to recycle ID's. Intuitively, this avoids the burstiness problem because only a *small fraction* of capabilities is revoked when system runs out of ID's. More precisely, each capability has a capability ID and a group ID. The disk keeps a list of valid group ID's, and for each valid group, a bitmap with the revocation state of ID's. To know if a capability is still valid, the disk checks if its group ID is valid and, within that group, whether the capability ID is not revoked. To recycle the capability ID's

pose that it was feasible for an adversary who does not know the value of k to produce values m, op, and c such that $m = h(op, h(c, k))$. Then by the unforgeability property of h mentioned above, the adversary must know the value of $h(c, k) = s$. Thus, the adversary can produce c and s such that $s = h(c, k)$. Therefore, by applying the unforgeability property again, the adversary must know the secret key k, a contradiction.

group ID	capability ID	disk ID	extents	mode

Figure 1: **Contents of a capability.** The group ID and capability ID are used in our new revocation scheme. The disk ID, extents, and mode describe the access granted by the capability.

2.1 The basic capability scheme

Our protocol for using capabilities is similar to that of NASD [10, 9] and SCARED [19], except that our capabilities describe access in terms of blocks rather than objects. Intuitively, a capability is a self-descriptive certificate that grants a specified type of access to parts of a disk (see Figure 1) when used with an associated secret. Our capabilities contain a group ID and a capability ID, which are used for revocations as explained in Section 2.2; a disk ID, which specifies the disk to which this capability applies; a list of *extents*, the ranges of physical blocks for which access is being granted; and an access mode (read, write, or both).

The secret is used to prevent forgery of illegal capabilities or of illegal requests using legal capabilities, as we now explain. The secret is generated using a *keyed-hash message authentication code*, or MAC [1]. A MAC function $h(\mathsf{data}, \mathsf{key})$ returns a string mac of fixed length with the following *unforgeability property*: without knowing the value of key, it is infeasible to find *any* new pairs of mac and data such that $\mathsf{mac} = h(\mathsf{data}, \mathsf{key})$. MAC functions can be computed quite efficiently in practice, unlike public-key signatures.

Every capability c is associated with a secret $h(c, k)$, where k is a secret key shared by the metadata server and the disk whose ID is specified in c. (There is a different key for each disk.) The use of this secret is best explained by an example. Figure 2 shows a client opening a file for the first time, and then reading or writing some data. To open the file, the client contacts the metadata server associated with the file. If the file's metadata is not cached at the server, the server must retrieve it from the relevant disk, shown by dashed lines in the figure (the metadata server accesses the disk in the same way that the client does, which we explain below).

The server checks if the client is permitted to access the file, and if so it gives the client the following: (1) the list of physical blocks comprising the file (its *blockmap*), (2) a capability c for the file's blocks[1] with the requested access mode (read, write, or both), and (3) the secret

[1]For simplicity, this example assumes that the file's blocks can be described using only one capability; in practice, highly fragmented files may require multiple capabilities because capabilities have a fixed size so that they can fit in a packet.

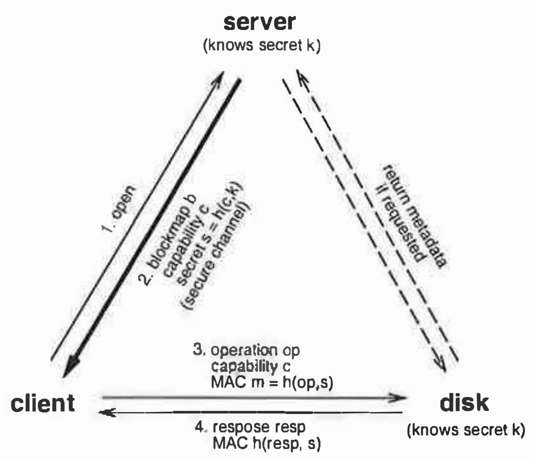


Figure 2: **Opening and accessing a file.** When a client wishes to access a file, it talks to the metadata server to get a capability c and its associated secret $s = h(c, k)$. The client can then access the file directly from disk. The disk verifies that the access is authentic by checking that the client has correctly computed the "double MAC" $m = h(op, s) = h(op, h(c, k))$.

$s = h(c, k)$ associated with c. The server's reply (in particular s) must be sent over a secure channel (shown by a darker line in the figure) to prevent eavesdroppers from learning the secret needed to use the capability. A secure channel can be obtained by encrypting under a block cipher using a session key established by an authentication protocol such as Kerberos.

Next, in order to read or write data from or to the file, the client issues block requests to the disk using the capability that it obtained. More precisely, the client sends to the disk an operation *op* that consists of (1) the type of access (read or write); (2) the range of physical blocks to be accessed; and (3) in case of a write operation, the data to be written. Together with *op*, the client also sends the capability c provided by the server and a MAC $m = h(op, s)$, where s is the secret associated with c. Because $m = h(op, s) = h(op, h(c, k))$, we call this trick the "double MAC". (The double MAC is not new; the earliest references we know of are Gobioff *et al.* [10] and Mittra *et al.* [16].) The disk can verify that the MAC is correct since it receives both c and *op*, and it has the secret key k. Note that the double MAC serves a double purpose: (1) it is a proof that the client knows s and thus has been authorized to use the capability c to issue the operation *op*, and (2) it prevents *op* from being tampered with, because if an attacker changes *op* it would not know how to compute the required new MAC.[2]

[2]The reader might be wondering whether the application of a MAC to its own output has somehow compromised its cryptographic properties. This is not the case, and the intuitive argument is as follows: Sup-

and key objects to keep track of which users can access each block—in essence, access control lists must be stored in a particular format so as to be understandable to the disk. SNAD also stores a small client signature (36-100 bytes) for each block, requiring it to either offer clients a non-power-of-two raw block size or suffer a substantial performance penalty.

We propose in this paper adding simple block-level security to network-attached disks. Our proposal maintains the existing NAD sequentially-numbered raw-block view of storage, allowing existing NAD clients to continue using their existing data layout and management strategies (*e.g.*, block-based backup); the only changes required are to create and pass along authorization information from the server through the clients to the disks. Using a raw-block view allows for maximum flexibility—higher-level primitives often limit what can be done. For example, it seems difficult to implement a file system that can take atomic snapshots on top of SNAD's built-in primitives.

By having NADs verify that a request comes unaltered from a client authorized to read or write that block, and by encrypting network traffic, we can provide a reasonable level of security even in an environment where end users control client machines and the network is vulnerable to attacks such as wiretapping and spoofing. Under our system, requests are authorized by an appropriate accompanying capability. Similarly to the NASD security approach [10], our central server, which we call a *metadata server*, issues and manages capabilities, setting policy, while the disks do only simple access checks.

Because the naive approach of one capability per block has too much overhead, each of our capabilities specifies access to one or more ranges of blocks (*extents*). Since we do not want our disks to have to understand which blocks belong to which files, we use *self-describing capabilities*: the accessible blocks and access mode can be determined from a capability without reference to any other data structures.

To allow revocation of capabilities, we require disks to remember which capabilities have been invalidated; this validity data must be held in RAM for speed reasons. An important contribution of our work is our revocation method based on *capability groups*, which dramatically compresses the validity data without sacrificing performance. Another contribution is our non-connection-based method of handling replay attacks, which allows for an unlimited number of clients while using only a small amount of memory.

Because of these techniques, our disk protocol's requirements on the NADs are very low: standard cryptographic functionality plus a small amount of RAM for capability management and replay detection (on the order of 128 KB). We believe this requirement is so small that the scheme could be deployed in existing NADs by simply changing their firmware, without modifying or adding hardware. By comparison, existing approaches to adding security to NADs require much deeper changes to the disks, and they would cost significantly more to implement.

The remainder of the paper is organized as follows: Section 2 describes our basic scheme to achieve security. Section 3 describes the implementation of a prototype NAD file system we built to demonstrate our scheme. Section 4 explains some other important aspects of our design. Section 5 discusses the prototype's performance, including the cost of security and the prototype's scalability. Section 6 covers some limitations of our design. Section 7 covers related work. And, finally, Section 8 concludes the paper.

2 Block-based security with modest RAM

We assume that the server and disks can be trusted—they are responsible for setting and enforcing security policies under our approach. However, we assume clients can be compromised and that the network is not secure, either from eavesdropping or spoofing. Under these conditions, our security ensures that attackers can access a user's data only by compromising a client machine logged into by that user. Should encryption be turned off for performance reasons, some privacy will be lost, as a wiretapper can see data read or written. The level of security we offer is similar to that of an NFS system that uses Kerberos for authentication, cryptographic checksums for integrity, and optional encryption for privacy.

Unforgeable, self-describing capabilities are the chief mechanism we employ for adding security to a NAD file system. We use the well-known idea of capabilities composed of two parts: a self-describing certificate and an associated secret. The secret is generated via a message authentication code (MAC) from the certificate and a hidden key known only to the server and the relevant disk [10, 9, 16, 19]. This basic capability approach, reviewed in Section 2.1 below, is augmented by two new techniques which permit RAM requirements on the system's disks to be very modest: a revocation scheme based on *capability groups*, described in Section 2.2; and a defense against replay attacks using Bloom filters, described in Section 2.4.

Block-Level Security for Network-Attached Disks

Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli,
Dave Andersen, Mike Burrows, Timothy Mann, Chandramohan A. Thekkath
*HP Systems Research Center**
Palo Alto, CA

Abstract

We propose a practical and efficient method for adding security to network-attached disks (NADs). In contrast to previous work, our design requires no changes to the data layout on disk, minimal changes to existing NADs, and only small changes to the standard protocol for accessing remote block-based devices. Thus, existing NAD file systems and storage-management software could incorporate our scheme very easily. Our design enforces security using the well-known idea of self-describing capabilities, with two novel features that limit the need for memory on secure NADs: a scheme to manage revocations based on *capability groups*, and a replay-detection method using Bloom filters.

We have implemented a prototype NAD file system, called *Snapdragon*, that incorporates our ideas. We evaluated Snapdragon's performance and scalability. The overhead of access control is small: latency for reads and writes increases by less than 0.5 ms (5%), while bandwidth decreases by up to 16%. The aggregate throughput scales linearly with the number of NADs (up to 7 in our experiments).

1 Introduction

Network-attached disks (NADs) are storage devices that accept block read/write requests over the network. They can be used to build file systems that provide better performance than traditional distributed file systems such as NFS [22]. In traditional systems, disks are attached directly to a file server, which provides file access to clients across a network. Because all data must pass through the server, it quickly becomes a bottleneck as the system scales, limiting the achievable bandwidth.

NAD file systems, in contrast, allow clients to bypass the file server and go straight to disk to read and write file data. Although clients must still talk to the file server for metadata operations (*e.g.*, file lookup and deletion), the file server's bandwidth no longer constrains the file system's bandwidth. Such *asymmetric shared file systems* [18] excel at workloads with high bandwidth and relatively few metadata operations. Commercial examples of such file systems include Tivoli's SANergy [25] and SGI's CXFS [11].

Network-attached disks commercially available today do not provide any support for security: they honor any request received. Thus, securing a NAD file system today requires the NAD network and all attached machines and disks to be physically secured. In practice, this forces the use of a separate LAN for storage (usually called a Storage Area Network, or SAN) and prevents clients located outside a machine room or under end-user control from directly accessing the NADs. Unfortunately, this means that NAD file systems cannot deliver high bandwidth to desktop machines, preventing many useful applications (*e.g.*, supplying training videos to PCs and desktop data mining) from taking advantage of NAD technology.

We believe that any practical approach to this problem should minimize the changes required in order for commercial NAD file systems to use it. Schemes that require major changes to commercial file systems, or the creation of a commercial-quality file system from scratch—both very expensive propositions—are unlikely to be adopted. Also of concern are the changes needed to storage-management software, such as monitoring, mirroring and backup tools.

Unfortunately, existing approaches that add security to NADs require large changes: NASD [7, 8] replaces the existing NAD block model with a variable-length data-object model and moves most of the filesystem functionality onto the disk. SUNDR [13] indexes blocks by their cryptographic hash instead of their (logical) position on the disk and garbage collects blocks whose *required-by* lists become empty. SNAD [5, 15] disks use special file

*This work was done at the HP Systems Research Center in Palo Alto. The authors with differing current affiliations are: Andersen, MIT; Burrows and Thekkath, Microsoft Research; Mann, VMware.

References

[1] M. Aboutabl, A. Agrawala, and J.-D. Decotignie. Temporally determinate disk access: An experimental approach. *Univ. of Maryland Technical Report CS-TR-3752*, 1997.

[2] R. T. Azuma. Tracking requirements for augmented reality. *Communications of the ACM*, 36(7):50–51, July 1993.

[3] E. Chang and H. Garcia-Molina. Bubbleup - Low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.

[4] S. J. Daigle and J. K. Strosnider. Disk scheduling for multimedia data streams. *Proceedings of the IS&T/SPIE*, February 1994.

[5] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Virtual IO: Preemptible disk access (poster). *Proceedings of the ACM Multimedia*, December 2002.

[6] Z. Dimitrijevic, R. Rangaswami, and E. Chang. The XTREAM multimedia system. *IEEE Conference on Multimedia and Expo*, August 2002.

[7] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya. Diskbench. *http://www.cs.ucsb.edu/~zoran/papers/db01.pdf*, November 2001.

[8] L. Huang and T. Chiueh. Implementation of a rotation-latency-sensitive disk scheduler. *SUNY at Stony Brook Technical Report*, May 2000.

[9] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, December 1991.

[10] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering*, 19(9):920–934, 1993.

[11] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *ACM Journal*, January 1973.

[12] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. *Proceedings of the Usenix FAST*, January 2002.

[13] C. R. Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle. Towards higher disk head utilization: Extracting free bandwith from busy disk drives. *Proceedings of the OSDI*, 2000.

[14] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing timing constraints for disk accesses in RT-Mach. *Proceedings of the Real Time Systems Symposium*, 1997.

[15] Performance Evaluation Laboratory, Brigham Young University. Trace distribution center. *http://tds.cs.byu.edu/tds/*, 2002.

[16] E. Riedel, C. Faloutsos, G. R. Ganger, and D. F. Nagle. Data mining on an OLTP system (nearly) for free. *Proceedings of the ACM SIGMOD*, May 2000.

[17] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 2:17–28, 1994.

[18] J. Schindler and G. R. Ganger. Automated disk drive characterization. *CMU Technical Report CMU-CS-00-176*, December 1999.

[19] Seagate Technology. Seagate's sound barrier technology. *http://www.seagate.com/docs/pdf/whitepaper/sound_barrier.pdf*, November 2000.

[20] C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri. On scheduling atomic and composite multimedia objects. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):447–455, 2002.

[21] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Proceedings of the ACM Sigmetrics*, June 1998.

[22] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based extraction of local and global disk characteristics. *UC Berkeley Technical Report*, 1999.

[23] W. Tavanapong, K. Hua, and J. Wang. A framework for supporting previewing and vcr operations in a low bandwidth environment. *Proceedings of the 5th ACM Multimedia Conference*, November 1997.

[24] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O—A novel I/O semantics for energy-aware applications. *Proceedings of the OSDI*, December 2002.

[25] B. L. Worthington, G. Ganger, Y. N. Patt, and J. Wilkes. Online extraction of scsi disk drive parameters. *Proceedings of the ACM Sigmetrics*, pages 146–156, 1995.

[26] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of the ACM Sigmetrics*, pages 241–251, May 1994.

[27] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading capacity for performance in a disk array. *Proceedings of the OSDI*, October 2000.

the average response time does not increase significantly with the increase in the arrival rate of higher-priority requests. However, the disk throughput does decrease with an increase in the arrival rate of higher-priority requests. As explained earlier, this reduction is expected since the overhead of IO preemption is an extra seek operation per preemption. For applications that require short response time, the performance penalty of IO preemption seems acceptable.

4.3.1 External Waiting Time

In Section 3.1, we explained the difference in the preemptibility of read and write IO requests and introduced the notion of external waiting time. Table 3 summarizes the effect of external waiting time on the preemption of write IO requests. The arrival rate of higher-priority requests is set to $\nu = 1$ req/s. As shown in Table 3, the average response time for higher-priority requests for write experiments is several times longer than for read experiments. Since the higher-priority requests have the same arrival pattern in both experiments, the average seek time and rotational delay are the same for both read and write experiments. The large and often unpredictable external waiting time in the write case explains these results.

	Exp. Waiting $[ms]$				*Avg. Response* $[ms]$			
IO	*npIO*		*spIO*		*npIO*		*spIO*	
$[kB]$	RD	WR	RD	WR	RD	WR	RD	WR
50	8.2	11.4	3.9	9.5	21.8	105.8	16.0	24.6
250	11.8	12.9	3.1	5.6	25.5	27.2	16.1	21.2
500	16.4	18.7	2.5	4.7	28.1	36.0	15.5	20.3

Table 3: The expected waiting time and average response time for non-preemptible and *Semi-preemptible IO* ($\nu = 1$ req/s).

Table 4 presents the results of our experiments aimed to find out the effect of write IO preemption on the average response time for higher-priority requests and disk write throughput. For example, in the case of 50 kB write IO requests, the disk can buffer multiple requests, and the write-back operation can include multiple seek operations. *Semi-preemptible IO* succeeds in reducing external waiting time and provides substantial improvement in the response time. However, since the disk is able to efficiently reorder the buffered write requests in the case of non-preemptible IO, it achieves better disk throughput. For large IO requests, *Semi-preemptible IO* achieves write throughput comparable to that of non-preemptible IO. We suggest that write preemption can be disabled when maintaining high system throughput is essential, and the disk reordering is useful (reordering could also be done in the operating system scheduler using the low-level disk knowledge).

IO	ν	*Avg. Response* $[ms]$		*Throughput* $[MB/s]$	
$[kB]$	$[req/s]$	*npIO*	*spIO*	*npIO*	*spIO*
50	*0.5*	93.1	26.9	4.85	1.98
50	*1*	105.8	24.6	4.75	1.96
50	*2*	91.1	22.7	4.68	1.94
50	*5*	102.2	24.4	4.40	1.84
50	*10*	87.5	23.7	3.95	1.70
50	*20*	81.3	23.3	3.09	1.42
500	*0.5*	32.4	20.3	13.71	11.41
500	*1*	36.0	20.3	13.64	11.24
500	*2*	35.0	20.8	13.45	11.02
500	*5*	34.9	20.5	12.82	10.36
500	*10*	36.6	20.3	11.67	9.13
500	*20*	34.6	20.7	9.64	6.92

Table 4: The average response time and disk write throughput for non-preemptible and *Semi-preemptible IO*.

5 Conclusion and Future Work

In this paper, we have presented the design of *Semi-preemptible IO*, and proposed three techniques for reducing IO waiting-time—data transfer chunking, just-in-time seek, and seek-splitting. These techniques enable the preemption of a disk IO request, and thus substantially reduce the waiting time for a competing higher-priority IO request. Using both synthetic and trace workloads, we have shown that these techniques can be efficiently implemented, given detailed disk parameters. Our empirical studies showed that *Semi-preemptible IO* can reduce the waiting time for both read and write requests significantly when compared with non-preemptible IOs.

We believe that preemptible IO can especially benefit multimedia and real-time systems, which are delay sensitive and which issue large-size IOs for meeting real-time constraints. We are currently implementing *Semi-preemptible IO* in Linux kernel. We plan to further study its performance impact on traditional and real-time disk-scheduling algorithms.

6 Acknowledgements

This project is partially supported by NSF grants IIS-0133802 (Career) and IIS-0219885 (ITR), and a UC DiMI grant. The third author is supported by an IBM Faculty Partnership Award. The authors wish to thank our reviewers and especially Erik Riedel from Seagate Research for their valuable comments and suggestions.

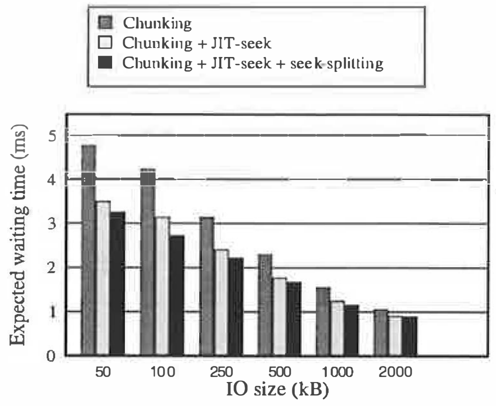


Figure 16: Individual contributions of *Semi-preemptible IO* components on the expected waiting time (Elevator).

transfer time dominates the seek and rotational delays, chunking is the most useful method for reducing the expected waiting time. When the seek and rotational delays are dominant, JIT-seek and seek-splitting become more effective for reducing the expected waiting time.

Figure 17 summarizes the individual contributions of the three strategies with respect to the achieved disk throughput. Seek-splitting can degrade disk throughput, since whenever a long seek is split, the disk requires more time to perform multiple sub-seeks. JIT-seek requires accurate prediction of the seek time and rotational delay. It introduces overhead in the case of mis-prediction. However, when the data transfer is dominant, benefits of chunking can mask both seek-splitting and JIT-seek overheads. JIT-seek aids the throughput with free prefetching. The potential free disk throughput acquired using free prefetching depends on the rate of JIT-seeks, which decreases with IO size. We believe

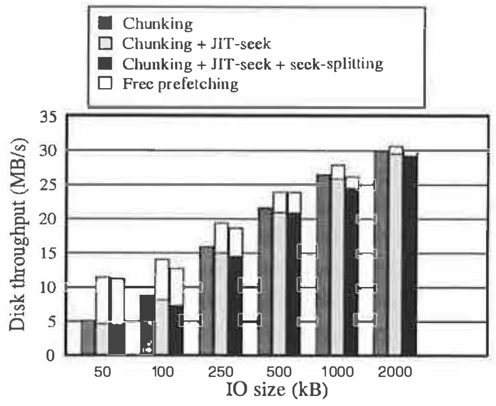


Figure 17: Individual effects of *Semi-preemptible IO* strategies on disk throughput (Elevator).

that the free prefetching is a useful strategy for multimedia systems that often access data sequentially and hence can use most of the potential free throughput.

4.3 Effect of Preemption

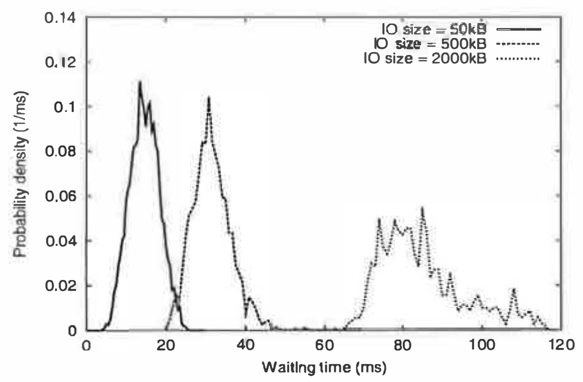
To estimate the response time for higher-priority IO requests, we conducted experiments wherein higher-priority requests were inserted into the IO queue at a constant rate (ν). While the constant arrival rate may seem unrealistic, the main purpose of this set of experiments is only to "estimate" the benefits and overheads associated with preempting an ongoing *Semi-preemptible IO* request to service a higher-priority IO request.

Table 2 presents the response time for a higher-priority request when using *Semi-preemptible IO* in two possible scenarios: (1) when the higher-priority request is serviced after the ongoing IO is completed (non-preemptible IO), and (2) when the ongoing IO is preempted to service the higher-priority IO request (*Semi-preemptible IO*). If the ongoing IO request is not preempted, then all higher-priority requests that arrive while it is being serviced, must wait until the IO is completed. The results in Table 2 illustrate the case when the ongoing request is a read request. The results for the write case are presented in Table 4.

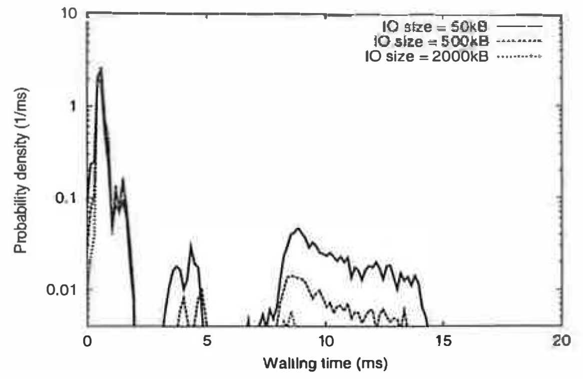
IO	ν	*Avg. Resp.* $[ms]$		*Throughput* $[MB/s]$	
$[kB]$	$[req/s]$	*npIO*	*spIO*	*npIO*	*spIO*
50	*0.5*	19.2	19.4	3.39	2.83
50	*1*	21.8	16.0	3.36	2.89
50	*2*	20.8	17.6	3.32	2.82
50	*5*	21.0	18.2	3.18	2.62
50	*10*	21.2	18.3	2.95	2.30
50	*20*	21.1	18.4	2.49	1.68
500	*0.5*	29.2	15.7	16.25	16.40
500	*1*	28.1	15.5	16.15	16.20
500	*2*	28.2	16.7	15.94	15.77
500	*5*	28.6	16.0	15.28	14.58
500	*10*	28.9	16.3	14.24	12.48
500	*20*	29.4	16.8	11.96	8.57

Table 2: The average response time and disk throughput for non-preemptible IO ($npIO$) and *Semi-preemptible IO* ($spIO$).

Preemption of IO requests is not possible without overhead. Each time a higher-priority request preempts a low-priority IO request for disk access, an extra seek is required to continue servicing the preempted request after the higher-priority request has been completed. Table 2 presents the average response time and the disk throughput for different arrival rates of higher-priority requests. For the same size of low-priority IO requests,



(a) Non-preemptible IO (linear scale)



(b) *Semi-preemptible IO* (logarithmic scale)

Figure 13: Distribution of the disk command duration (FCFS). Smaller values imply a higher preemptibility.

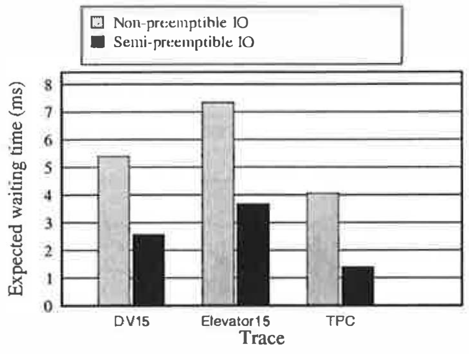


Figure 14: Improvement in the expected waiting time (using disk traces).

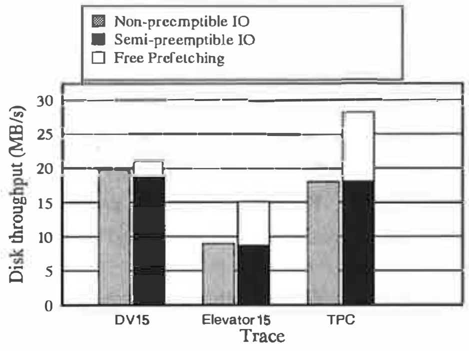


Figure 15: Effect on the achieved disk throughput (using disk traces).

from three applications. The first trace (DV15) was obtained when the XTREAM multimedia system [6] was servicing 15 simultaneous video clients using the FCFS disk scheduler. The second trace (Elevator15) was obtained using the similar setup where XTREAM let Linux elevator scheduler handle concurrent disk IOs. The third was a disk trace of the TPC-C database benchmark with 20 warehouses obtained from [15]. Trace summary is presented in Table 1.

Trace	Number of requests	Avg. req. size [*blocks*]	Max. block number
DV15	10800	128.7	28442272
Elevator15	10180	127.6	28429968
TPC	1376482	126.5	8005312

Table 1: Trace summary.

Figures 14 and 15 show the expected waiting time and disk throughput for these trace experiments. The expected waiting time was reduced by as much as 65% (Figure 14) with less than 10% (Figure 15) loss in disk throughput for all traces. (Elevator15 had smaller throughput than DV15 because several processes were accessing the disk concurrently, which increased the total number of seeks.)

4.2 Individual Contributions

Figure 16 shows the individual contributions of the three strategies with respect to expected waiting time for the random workload with the elevator scheduling policy. In Section 4.1, we showed that the expected waiting time can be significantly smaller in *Semi-preemptible IO* than in non-preemptible IO. Here we compare only contributions within *Semi-preemptible IO* to show the importance of each strategy. Since the time to transfer a single chunk of data is small compared to the seek time (typically less than 1 ms for a chunk transfer and 10 ms for a seek), the expected waiting time decreases as the data transfer time becomes more dominant. When the data

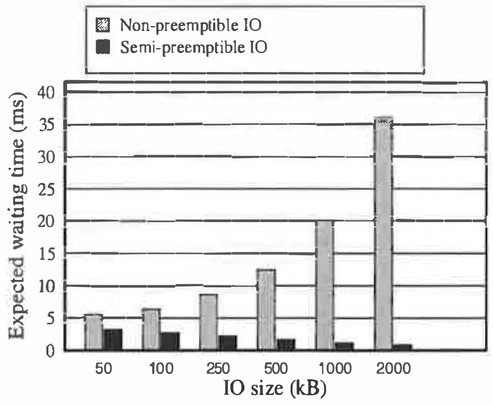


Figure 10: Improvements in the expected waiting time (Elevator).

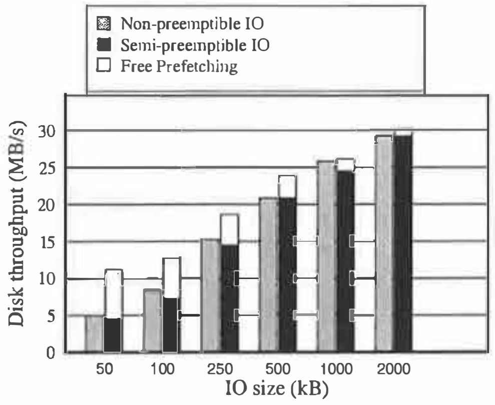


Figure 12: Effect on achieved disk throughput (Elevator).

able but minor reduction in disk throughput using *Semi-preemptible IO* (less than 15%). This reduction is due to the overhead of seek-splitting and mis-prediction of seek and rotational delay. More details on the accuracy of rotational delay predictions can be found in [7]. Another point worth mentioning is that the reduction in disk throughput in *Semi-preemptible IO* is smaller for large IOs than for small IOs due to the reduced number of seeks and hence the smaller overhead.

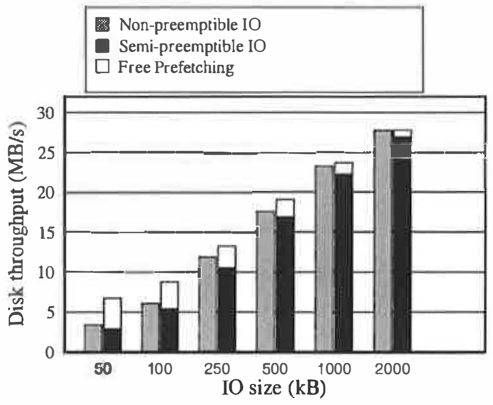


Figure 11: Effect on achieved disk throughput (FCFS).

Since disk commands are non-preemptible (even in *Semi-preemptible IO*), we can use the duration of disk commands to measure the expected waiting time. A smaller value implies a more preemptible system. Figure 13 shows the distribution of the durations of disk commands for both non-preemptible IO and *Semi-preemptible IO* (for exactly the same sequence of IO requests). In the case of non-preemptible IO (Figure 13 (a)), one IO request is serviced using a single disk command. Hence, the disk access can be preempted only when the current IO request is completed. The distribution is dense near the sum of the average seek time, rotational delay, and transfer time required to service an entire IO request. The distribution is wider when the IO requests are larger, because the duration of data transfer depends not only on the size of the IO request, but also on the throughput of the disk zone where the data resides.

In the case of *Semi-preemptible IO*, the distribution of the durations of disk commands does not directly depend on the IO request size, but on individual disk commands used to perform an IO request. (We plot the distribution for the *Semi-preemptible IO* case in logarithmic scale, so that the probability density of longer disk commands can be better visualized.) In Figure 13 (b), we see that for *Semi-preemptible IO*, the largest probability density is around the time required to transfer a single chunk of data. If the chunk includes the track or cylinder skew, the duration of the command will be slightly longer. (The two peaks immediately to the right of the highest peak, at approximately 2 ms, have the same probability because the disk used in our experiments has two tracks per cylinder.) The part of the distribution between 3 ms and 16 ms in the figure is due to the combined effect of JIT-seek and seek-splitting on the seek and rotational delays. The probability for this range is small, approximately 0.168, 0.056, and 0.017 for 50 kB, 500 kB, and 2,000 kB IO requests, respectively.

4.1.2 Trace Workload

We now present preemptibility results using IO traces obtained from a Linux system. IO traces were obtained

- What is the effect of IO *preemption* on the average response time for higher-priority requests and the disk throughput?

In order to answer these questions, we have implemented a prototype system which can service IO requests using either the traditional non-preemptible method (*non-preemptible IO*) or *Semi-preemptible IO*. Our prototype runs as a user-level process in Linux and talks directly to a SCSI disk using the Linux SCSI-generic interface. The prototype uses the logical-to-physical block mapping of the disk, the seek curves, and the rotational skew times, all of which are automatically generated by the Diskbench [7]. All experiments were performed on a Pentium III 800 MHz machine with a Seagate ST318437LW SCSI disk. This SCSI disk has two tracks per cylinder, with 437 to 750 blocks per track depending on the disk zone. The total disk capacity is 18.4 GB. The rotational speed of the disk is 7200 RPM. The maximum sequential disk throughput is between 24.3 and 41.7 MBps.

For performance benchmarking, we performed two sets of experiments. First, we tested the preemptibility of the system using simulated IO workload. For the simulated workload, we used equal-sized IO requests within each experiment. The low-priority IOs are for data located at random positions on the disk. In the experiments where we actually performed preemption, the higher-priority IO requests were also at random positions. However, their size was set to only one block in order to provide the lower estimate for preemption overhead. We tested the preemptibility under *first-come-first-serve (FCFS)* and *elevator* disk scheduling policies. In the second set of experiments we used trace workload obtained on the tested Linux system. We obtained the traces from the instrumented Linux-kernel disk-driver. In the simulated experiments, non-preemptible IOs are serviced using chunk sizes of 128 kB. This is the size used by Linux and FreeBSD for breaking up large IOs. We assume that a large IO cannot be preempted between chunks, since such is the case for current operating systems. On the other hand, our prototype services larger IOs using multiple disk commands and preemption is possible after each disk command is completed. Based on disk profiling, our prototype used the following parameters for *Semi-preemptible IO*. Chunking divided the data transfer into chunks of 50 disk blocks each, except for the last chunk, which can be smaller. JIT-seek used an offset of 1 ms to reduce the probability of prediction errors. Seeks for more than a half of the disk size in cylinders were split into two equal-sized, smaller seeks. We used the SCSI *seek* command to perform sub-seeks.

4.1 Preemptibility

The experiments for preemptibility of disk access measure the duration of (non-preemptible) disk commands in both non-preemptible IO and *Semi-preemptible IO* in the absence of higher-priority IO requests. The results include both detailed distribution of disk commands durations (and hence maximum possible waiting time) and the expected waiting time calculated using Equations 1 and 2, as explained in Section 3.

4.1.1 Random Workload

Figure 9 depicts the difference in the expected waiting time between non-preemptible IO and *Semi-preemptible IO*. In this experiment, IOs were serviced for data situated at random locations on the disk. The IOs were serviced using FCFS policy. We can see that the expected waiting time for non-preemptible IOs increases linearly with IO size due to increased data transfer time. However, the expected waiting time for *Semi-preemptible IO* actually decreases with IO size, since the disk spends more time in data transfer, which is more preemptible.

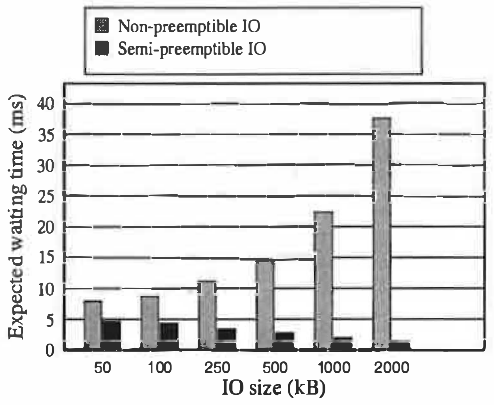
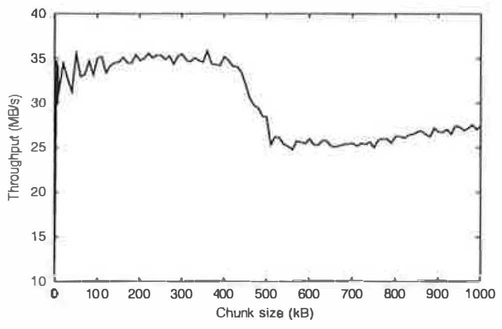


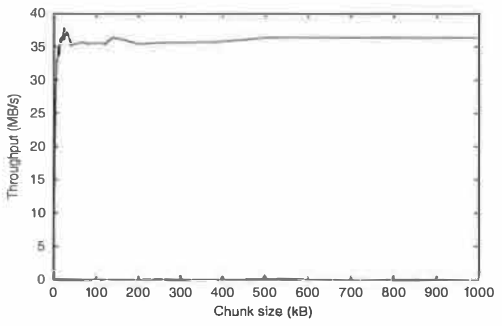
Figure 9: Improvements in the expected waiting time (FCFS).

Figure 10 depicts the improvements in the expected waiting time when the system uses an elevator-based scheduling policy. (The figure shows the results of randomly generated IO requests serviced in batches of 40.) The results are better than those of FCFS access since the elevator scheduler reduces the seek component that is the least-preemptible.

Figures 11 and 12 show the effect of improving IO preemptibility on the achieved disk throughput when an FCFS scheduling policy is used. There is a notice-



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 6: Sequential write throughput vs. chunk size.

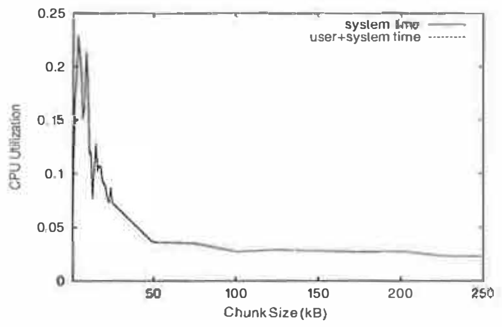


Figure 7: CPU utilization vs. chunk size for IDE WD400BB.

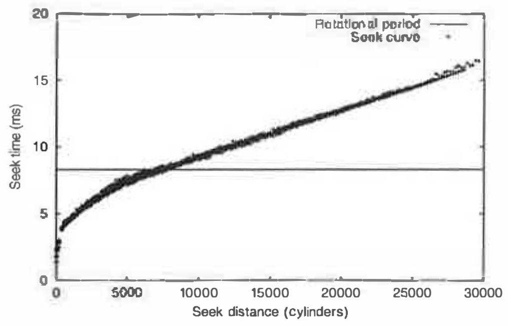


Figure 8: Seek curve for SCSI ST318437LW.

one SCSI and one IDE disk drive. Figure 6 shows the same for the write case. Clearly, the optimal range for the chunk size (between the points a and b illustrated previously in Figure 3) can be automatically extracted from these figures. The disk profiler implementation was successful in extracting the optimal chunk size for several SCSI and IDE disk drives with which we experimented. For those who might also be interested in the CPU overhead for performing chunking, we present the CPU utilization when transferring a large data segment from the disk, using different chunk sizes in Figure 7 for an IDE disk. The CPU utilization decreases rapidly with an increase in the chunk size. Beyond a chunk size of 50 kB, the CPU utilization remains relatively constant. This figure shows that chunking, using even small chunk size (50 kB), is feasible for IDE disk without incurring any significant CPU overhead. For SCSI disks, the CPU overhead of chunking is even less than that for IDE disks, since the bulk of the processing is done by the SCSI controller.

To perform JIT-seek, the system needs an accurate estimate of the seek delay between two disk blocks. The disk profiler provides the seek curve as well as the variations in seek time. The seek time curve (and variations in seek time) for a SCSI disk obtained by the disk profiler is presented in Figure 8. The disk profiler also obtains the required parameters for rotational delay prediction between accessing two disk blocks in succession with near-microsecond-level precision. However, the variations in seek time can be of the order of one millisecond, which restricts the possible accuracy of prediction. Finally, to perform JIT-seek, the system combines seek time and rotational delay prediction to predict T_{rot}. We have conducted more detailed study on T_{rot} prediction in [7].

4 Experimental Results

We now present the performance results for our implementation of *Semi-preemptible IO*. Our experiments aimed to answer the following questions:

- What is the level of *preemptibility* of *Semi-preemptible IO* and how does it influence the disk throughput?
- What are the *individual contributions* of the three components of *Semi-preemptible IO*?

Benefits: The *seek-splitting* method reduces the T_{seek} component of the waiting time. A long non-preemptible seek can be transformed into multiple shorter sub-seeks. A higher-priority request can now be serviced at the end of a sub-seek, instead of being delayed until the entire seek operation is finished. For example, suppose an IO request involves a seek of $20,000$ cylinders, requiring a T_{seek} of 14 ms. Using seek-splitting, this seek operation can be divided into two 9 ms sub-seeks of $10,000$ cylinders each. Then the expected waiting time for a higher-priority request is reduced from 7 ms to 4.5 ms.

Overhead:

1. Due to the mechanics of the disk arm, the total time required to perform multiple sub-seeks is greater than that for a single seek of a given seek distance. Thus, the seek-splitting method can degrade disk throughput. Later in this section, we discuss this issue further.
2. Splitting the seek into multiple sub-seeks increases the number of disk head accelerations and decelerations, consequently increasing the power usage and noise.

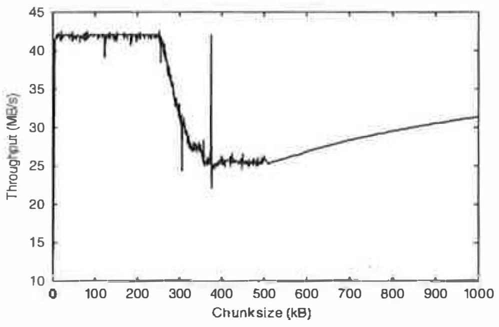
3.3.1 The Method

To split seek operations, *Semi-preemptible IO* uses a tunable parameter, the maximum sub-seek distance. The *maximum sub-seek distance* decides whether to split a seek operation. For seek distances smaller than the maximum sub-seek distance, seek-splitting is not employed. A smaller value for the maximum sub-seek distance provides higher responsiveness at the cost of possible throughput degradation.

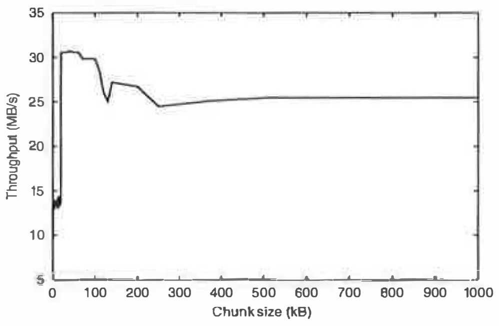
Unlike the previous two methods, seek-splitting may degrade disk performance. However, we note that the overhead due to seek-splitting can, in some cases, be masked. If the pre-seek slack obtained due to JIT-seek is greater than the seek overhead, then the slack can be used to mask this overhead. A specific example of this phenomenon was presented in Section 1. If the slack is insufficient to mask the overhead, seek-splitting can be aborted to avoid throughput degradation. Making such a tradeoff, of course, depends on the requirements of the application.

3.4 Disk Profiling

As mentioned in the beginning of this section, *Semi-preemptible IO* greatly relies on disk profiling to obtain accurate disk parameters. The disk profiler obtains the following required disk parameters:



(a) SCSI ST318437LW



(b) IDE WD400BB

Figure 5: Sequential read throughput vs. chunk size.

- **Disk block mappings.** System uses disk mappings for both logical-to-physical and physical-to-logical disk block address transformation.
- **The optimal chunk size.** In order to efficiently perform chunking, *Semi-preemptible IO* chooses the optimal chunk size from the optimal range extracted using disk profiler.
- **Disk rotational factors.** In order to perform JIT-seek, system requires accurate rotational delay prediction, which relies on disk rotation period and rotational skew factors for disk tracks.
- **Seek curve.** JIT-seek and seek-splitting methods rely on accurate seek time prediction.

The extraction of these disk parameters is described in [7].

As regards chunking, the disk profiler provides the optimal range for the chunk size. Figure 5 depicts the effect of chunk size on the read throughput performance for in SCSI write command. Using this simple technique, it triggers the write-back operation at the end of each write IO. Consequently, the external waiting time is reduced since the write-back operation does not include multiple disk seeks.

3.2 JIT-seek: Preempting $\mathbf{T_{rot}}$

After the reduction of the $T_{transfer}$ component of the waiting time, the rotational delay and seek time components become significant. The rotational period (T_P) can be as much as 10 ms in current-day disk drives. To reduce the rotational delay component (T_{rot}) of the waiting time, we propose a *just-in-time seek* (*JIT-seek*) technique for IO operations.

Definition 3.2: The *JIT-seek* technique delays the servicing of the next IO request in such a way that the rotational delay to be incurred is minimized. We refer to the delay between two IO requests, due to JIT-seek, as *slack time*.

Benefits:

1. The slack time between two IO requests is fully preemptible. For example, suppose that an IO request must incur a T_{rot} of 5 ms, and JIT-seek delays the issuing of the disk command by 4 ms. The disk is thus idle for $T_{idle} = 4$ ms. Then, the expected waiting time is reduced from 2.5 ms to $\frac{1}{2}\frac{1 \times 1}{1+4} = 0.1$ ms.

2. The slack obtained due to JIT-seek can also be used to perform data prefetching for the previous IO or to service a background request, and hence potentially increase the disk throughput.

Overhead: *Semi-preemptible IO* predicts the rotational delay and seek time between two IO operations in order to perform JIT-seek. If there is an error in prediction, then the penalty for JIT-seek is at most one extra disk rotation and some wasted cache space for unused prefetched data.

3.2.1 The Method

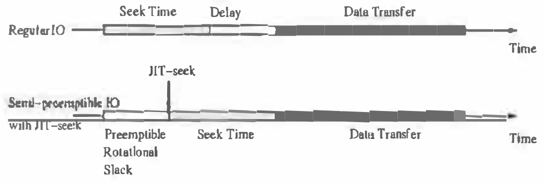


Figure 4: JIT-seek.

The JIT-seek method is illustrated in Figure 4. The x-axis depicts time, and the two horizontal lines depict a regular IO and an IO with JIT-seek, respectively. With JIT-seek, the read command for an IO operation is delayed and issued just-in-time so that the seek operation takes the disk head directly to the destination block, without incurring any rotational delay at the destination track. Hence, data transfer immediately follows the seek operation. The available rotational slack, before issuing the JIT-seek command, is now preemptible. We can make two key observations about the JIT-seek method. First, an accurate JIT-seek operation reduces the T_{rot} component of the waiting time without any loss in performance. Second, and perhaps more significantly, the ongoing IO request can be serviced as much as possible, or even completely, if sufficient slack is available before the JIT-seek operation for a higher-priority request.

The pre-seek slack made available due to the JIT-seek operation can be used in three possible ways:

- The pree-seek slack can be simply left unused. In this case, a higher-priority request arriving during the slack time can be serviced immediately.
- The slack can be used to perform additional data transfers. Operating systems can perform data prefetching for the current IO beyond the necessary data transfer. We refer to it as *free prefetching* [13]. Chunking is used for the prefetched data, to reduce the waiting time of a higher-priority request. Free prefetching can increase the disk throughput. We must point out, however, that free prefetching is useful only for sequential data streams where the prefetched data will be consumed within a short time. Operating systems can also perform another background request as proposed elsewhere [13, 16].
- The slack can be used to mask the overhead incurred in performing *seek-splitting*, which we shall discuss next.

3.3 Seek Splitting: Preempting $\mathbf{T_{seek}}$

The seek delay (T_{seek}) becomes the dominant component when the $T_{transfer}$ and T_{rot} components are reduced drastically. A full stroke of the disk arm may require as much as 20 ms in current-day disk drives. It may then be necessary to reduce the T_{seek} component to further reduce the waiting time.

Definition 3.3: *Seek-splitting* breaks a long, non-preemptible seek of the disk arm into multiple smaller sub-seeks.

3.1.1 The Method

To perform chunking, the system must decide on the chunk size. *Semi-preemptible IO* chooses the minimum chunk size for which the disk throughput is optimal and the CPU overhead acceptable. Surprisingly, large chunk sizes can also suffer from throughput degradation due to the sub-optimal implementation of disk firmware (Section 3.4). Consequently, *Semi-preemptible IO* may achieve even better disk throughput than the traditional method where an IO request is serviced using a single disk command.

In order to perform chunking efficiently, *Semi-preemptible IO* relies on the existence of a read cache and a write buffer on the disk. It uses disk profiling to find the optimal chunk size. We now present the chunking for read and write IO requests separately.

3.1.2 The Read Case

Disk drives are optimized for sequential access, and they continue prefetching data into the disk cache even after a read operation is completed [17]. Chunking for a read IO requests is illustrated in Figure 2. The x-axis shows time, and the two horizontal time lines depict the activity on the IO bus and the disk head, respectively. Employing chunking, a large $T_{transfer}$ is divided into smaller chunk transfers issued in succession. The first read command issued on the IO bus is for the first chunk. Due to the prefetching mechanism, all chunk transfers following the first one are serviced from the disk cache rather than the disk media. Thus, the data transfers on the IO bus (the small dark bars shown on the IO bus line in the figure) and the data transfer into the disk cache (the dark shaded bar on the disk-head line in the figure) occur concurrently. The disk head continuously transfers data after the first read command, thereby fully utilizing the disk throughput.

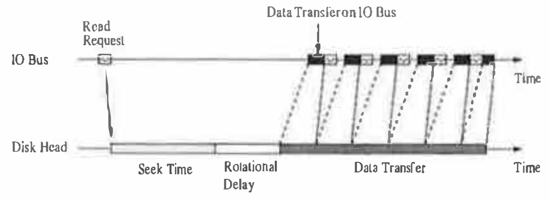


Figure 2: Virtual preemption of the data transfer.

Figure 3 illustrates the effect of the chunk size on the disk throughput using a mock disk. The optimal chunk size lies between a and b. A smaller chunk size reduces the waiting time for a higher-priority request. Hence, *Semi-preemptible IO* uses a chunk size close to but larger than a. For chunk sizes smaller than a, due to the overhead associated with issuing a disk command, the IO bus is a bottleneck. Point b in Figure 3 denotes the point beyond which the performance of the cache may be sub-optimal[3].

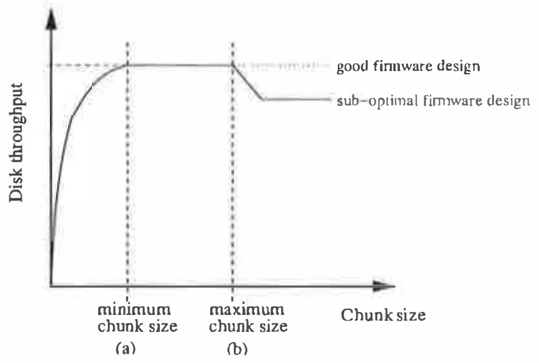


Figure 3: Effect of chunk size on disk throughput.

3.1.3 The Write Case

Semi-preemptible IO performs chunking for write IOs similarly to chunking for read requests. However, the implications of chunking in the write case are different. When a write IO is performed, the disk command can complete as soon as all the data is transferred to the disk write buffer[4]. As soon as the write command is completed, the operating system can issue a disk command to service a higher-priority IO. However, the disk may choose to schedule a write-back operation for disk write buffers before servicing a new disk command.We refer to this delay as the *external waiting time*. Since the disk can buffer multiple write requests, the write-back operation can include multiple disk seeks. Consequently, the waiting time for a higher-priority request can be substantially increased when the disk services write IOs.

In order to increase preemptibility of write requests, we must take into consideration the external waiting time for write IO requests. External waiting can be reduced to zero by disabling write buffering. However, in the absence of write buffering, chunking would severely degrade disk performance. The disk would suffer from an overhead of one disk rotation after performing an IO for each chunk. To remedy external waiting, our prototype forces the disk to write only the last chunk of the write IO to disk media by setting force-unit-access flag

[3]We have not fully investigated the reasons for sub-optimal disk performance and it is the subject of our future work.

[4]If the size of the write IO is larger than the size of the write buffer, then the disk signals the end of the IO as soon as the excess amount of data (which cannot be fitted into the disk buffer) has been written to the disk media.

- The *waiting time* is the time between the arrival of a higher-priority IO request and the moment the disk starts servicing it.

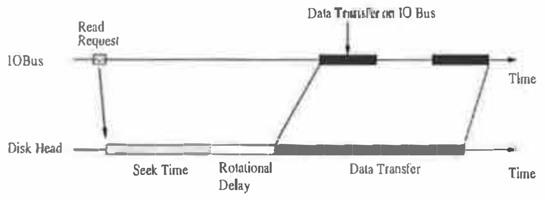


Figure 1: Timing diagram for a disk read request.

In order to understand the magnitude of the waiting time, let us consider a typical read IO request, depicted in Figure 1. The disk first performs a seek to the destination cylinder requiring T_{seek} time. Then, the disk must wait for a rotational delay, denoted by T_{rot}, so that the target disk block comes under the disk arm. The final stage is the data transfer stage, requiring a time of $T_{transfer}$, when the data is read from the disk media to the disk buffer. This data is simultaneously transferred over the IO bus to the system memory.

For a typical commodity system, once a disk command is issued on the IO bus, it cannot be stopped. Traditionally, an IO request is serviced using a single disk command. Consequently, the operating system must wait until the ongoing IO is completed before it can service the next IO request on the same disk. Let us assume that a higher-priority request may arrive at any time during the execution of an ongoing IO request with equal probability. The waiting time for the higher-priority request can be as long as the duration of the ongoing IO. The expected waiting time of a higher-priority IO request can then be expressed in terms of seek time, rotational delay, and data transfer time required for ongoing IO request as

$$E(T_{waiting}) = \frac{1}{2}(T_{seek} + T_{rot} + T_{transfer}). \quad (1)$$

Let V_i be the sequence of fine-grained disk commands we use to service an IO request. Let the time required to execute disk-command V_i be T_i. Let T_{idle} be the duration of time during the servicing of the IO request, when the disk is idle (i.e., no disk command is issued). Using the above assumption that the higher-priority request can arrive at any time with equal probability, the probability that it will arrive during the execution of the i^{th} command V_i can be expressed as $p_i = \frac{T_i}{\sum T_i + T_{idle}}$. Finally, the expected waiting time of a higher-priority request in *Semi-preemptible IO* can be expressed as

$$E(T'_{waiting}) = \frac{1}{2}\sum(p_i T_i) = \frac{1}{2}\frac{\sum T_i^2}{(\sum T_i + T_{idle})}. \quad (2)$$

In the remainder of this section, we present 1) *chunking*, which divides $T_{transfer}$ (Section 3.1); 2) *just-in-time seek*, which enables T_{rot} preemption (Section 3.2); and 3) *seek splitting*, which divides T_{seek} (Section 3.3). In addition, we present our disk profiler, Diskbench, and summarize all the disk parameters required for the implementation of *Semi-preemptible IO* (Section 3.4).

3.1 Chunking: Preempting $\mathbf{T_{transfer}}$

The data transfer component ($T_{transfer}$) in disk IOs can be large. For example, the current maximum disk IO size used by Linux and FreeBSD is 128 kB, and it can be larger for some specialized video-on-demand systems[2]. To make the $T_{transfer}$ component preemptible, *Semi-preemptible IO* uses *chunking*.

Definition 3.1: *Chunking* is a method for splitting the data transfer component of an IO request into multiple smaller *chunk* transfers. The chunk transfers are serviced using separate disk commands, issued sequentially.

Benefits: Chunking reduces the transfer component of $T_{waiting}$. A higher-priority request can be serviced after a chunk transfer is completed instead of after the entire IO is completed. For example, suppose a 500 kB IO request requires a $T_{transfer}$ of 25 ms at a transfer rate of 20 MBps. Using a chunk size of 20 kB, the expected waiting time for a higher-priority request is reduced from 12.5 ms to 0.5 ms.

Overhead: For small chunk sizes, the IO bus can become a performance bottleneck due to the overhead of issuing a large number of disk commands. As a result, the disk throughput degrades. Issuing multiple disk commands instead of a single one also increases the CPU overhead for performing IO. However, for the range of chunk sizes, the disk throughput using chunking is optimal with negligible CPU overhead.

[2]These values are likely to vary in the future. *Semi-preemptible IO* provides a technique that does not deter disk preemptibility with the increased IO sizes.

reduce the waiting time for a higher-priority request at little or no extra cost.

- We show that making write IOs preemptible is not as straightforward as it is for read IOs. We propose one possible solution for making them preemptible.
- We present a feasible path to implement *Semi-preemptible IO*. We explain how the implementation is made possible through use of a detailed disk profiling tool.

The rest of this paper is organized as follows: Section 2 presents related research. Section 3 introduces *Semi-preemptible IO* and describes its three components. In Section 4, we evaluate our prototype. In Section 5, we make concluding remarks and suggest directions for future work.

2 Related Work

Before the pioneering work of [4, 14], it was assumed that the nature of disk IOs was inherently non-preemptible. In [4], the authors proposed breaking up a large IO into multiple smaller chunks to reduce the data transfer component ($T_{transfer}$) of the *waiting time* ($T_{waiting}$) for higher-priority requests. A minimum chunk size of one track was proposed. In this paper, we improve upon the conceptual model of [4] in three respects: 1) in addition to enabling preemption of the data transfer component, we show how to enable preemption of T_{rot} and T_{seek} components; 2) we improve upon the bounds for zero-overhead preemptibility; and 3) we show that making write IOs preemptible is not as straightforward as it is for read IOs, but we propose one possible solution.

Weissel et al. [24] recently proposed Cooperative I/O, a novel IO semantics aimed to reduce the power consumption of storage subsystem by enabling applications to provide more information to OS scheduler. Similarly, in this paper we propose an IO abstraction to enable preemptive disk scheduling.

Semi-preemptible IO uses a *just-in-time seek* (JIT-seek) technique to make the rotational delay preemptible. JIT-seek can also be used to mask the rotational delay with useful data prefetching. In order to implement both methods, our system relies on accurate disk profiling [1, 7, 18, 22, 25]. Rotational delay masking has been proposed in multiple forms. In [8, 26], the authors present rotational-latency-sensitive schedulers, which consider the rotational position of the disk arm to make better scheduling decisions. In [13, 16, 12], the authors present *freeblock scheduling*, wherein the disk arm services background jobs using the rotational delay between foreground jobs. In [19], Seagate uses a variant of just-in-time seek in some of its disk drives to reduce power consumption and noise. *Semi-preemptible IO* uses similar techniques for a different goal—to make rotational delays preemptible.

There is a large body of literature proposing IO scheduling policies for multimedia and real-time systems that improve disk response time [3, 20, 21, 23]. *Semi-preemptible IO* is orthogonal to these contributions. We believe that the existing methods can benefit from using preemptible IO to improve schedulability and further decrease response time for higher-priority requests. For instance, to model real-time disk IOs, one can draw from real-time CPU scheduling theory. In [14], the authors adapt the *Earliest Deadline First* (EDF) algorithm from CPU scheduling to disk IO scheduling. Since EDF is a preemptive scheduling algorithm, a higher-priority request must be able to preempt a lower-priority request. However, an ongoing disk request cannot be preempted instantaneously. Applying such classical real-time CPU scheduling theory is simplified if the preemption granularity is independent of system variables like IO sizes. *Semi-preemptible IO* provides such an ability.

3 Semi-preemptible IO

Before introducing the concept of *Semi-preemptible IO*, we first define some terms which we will use throughout the rest of this paper. Then, we propose an abstraction for disk IO, which enables preemption of IO requests. Finally, we present our disk profiler and the disk parameters required for the implementation of *Semi-preemptible IO*.

Definitions:

- A *logical disk block* is the smallest unit of data that can be accessed on a disk drive (typically 512 B). Each logical block resides at a physical disk location, depicted by a physical address (cylinder, track, sector).
- A *disk command* is a non-preemptible request issued to the disk over the IO bus (e.g., the read, write, seek, and interrogative commands).
- An *IO request* is a request for read or write access to a sequential set of logical disk blocks.

thus separates the preemptibility from the size and duration of the operating system's IO requests.

Semi-preemptible IO maps each IO request into multiple fast-executing disk commands using three methods. Each method addresses the reduction of one of the possible components of the waiting time—ongoing IO's transfer time ($T_{transfer}$), rotational delay (T_{rot}), and seek time (T_{seek}).

- **Chunking $\mathbf{T_{transfer}}$.** A large IO transfer is divided into a number of small chunk transfers, and preemption is made possible between the small transfers. If the IO is not preempted between the chunk transfers, chunking does not incur any overhead. This is due to the prefetching mechanism in current disk drives (Section 3.1).
- **Preempting $\mathbf{T_{rot}}$.** By performing just-in-time (JIT) seek for servicing an IO request, the rotational delay at the destination track is virtually eliminated. The pre-seek slack time thus obtained is preemptible. This slack can also be used to perform prefetching for the ongoing IO request, or/and to perform seek splitting (Section 3.2).
- **Splitting $\mathbf{T_{seek}}$.** *Semi-preemptible IO* can split a long seek into sub-seeks, and permits a preemption between two sub-seeks (Section 3.3).

The following example illustrates how *Semi-preemptible IO* can reduce the waiting time for higher-priority IOs (and hence improve the preemptibility of disk access).

1.1 Illustrative Example

Suppose a 500 kB read-request has to seek $20,000$ cylinders requiring T_{seek} of 14 ms, must wait for a T_{rot} of 7 ms, and requires $T_{transfer}$ of 25 ms at a transfer rate of 20 MBps. The expected waiting time, $E(T_{waiting})$, for a higher-priority request arriving during the execution of this request, is 23 ms, while the maximum waiting time is 46 ms (please refer to Section 3 for equations). *Semi-preemptible IO* can reduce the waiting time by performing the following operations.

It first predicts both the seek time and rotational delay. Since the predicted seek time is long ($T_{seek} = 14$ ms), it decides to split the seek operation into two sub-seeks, each of $10,000$ cylinders, requiring $T'_{seek} = 9$ ms each. This seek splitting does not cause extra overhead in this case because the $T_{rot} = 7$ can mask the 4 ms increased total seek time ($2 \times T'_{seek} - T_{seek} = 2 \times 9 - 14 = 4$). The rotational delay is now $T'_{rot} = T_{rot} - (2 \times T'_{seek} - T_{seek}) = 3$ ms.

With this knowledge, the disk driver waits for 3 ms before performing a JIT-seek. This JIT-seek method makes T'_{rot} preemptible, since no disk operation is being performed. The disk then performs the two sub-seek disk commands, and then 25 successive read commands, each of size 20 kB, requiring 1 ms each. A higher-priority IO request could be serviced immediately after each disk-command. *Semi-preemptible IO* thus enables preemption of an originally non-preemptible read IO request. Now, during the service of this IO, we have two scenarios:

- **No higher-priority IO arrives.** In this case, the disk does not incur additional overhead for transferring data due to disk prefetching (discussed in Sections 3.1 and 3.4). (If T_{rot} cannot mask seek-splitting, the system can also choose not to perform seek-splitting.)
- **A higher-priority IO arrives.** In this case, the maximum waiting time for the higher-priority request is now a mere 9 ms, if it arrives during one of the two seek disk commands. However, if the ongoing request is at the stage of transferring data, the longest stall for the higher-priority request is just 1 ms. The expected value for waiting time is only $\frac{1}{2}\frac{2\times 9^2+25\times 1^2}{2\times 9+25\times 1+3} = 2.03$ ms, a significant reduction from 23 ms (refer to Section 3 for details).

This example shows that *Semi-preemptible IO* substantially reduces the expected waiting time and hence increases the preemptibility of disk access. However, if an IO request is preempted to service a higher-priority request, an extra seek operation may be required to resume service for the preempted IO. The distinction between *IO preemptibility* and *IO preemption* is an important one. Preemptibility enables preemption, but incurs little overhead itself. Preemption will always incur overhead, but it will reduce the service time for higher-priority requests. Preemptibility provides the system with the choice of trading throughput for short response time when such a tradeoff is desirable. We explore the effects of IO preemption further, in Section 4.3.

1.2 Contributions

In summary, the contributions of this paper are as follows:

- We introduce *Semi-preemptible IO*, which abstracts both read and write IO requests so as to make them preemptible. As a result, system can substantially

Design and Implementation of Semi-preemptible IO

Zoran Dimitrijević Raju Rangaswami Edward Chang
University of California, Santa Barbara
zoran@cs.ucsb.edu raju@cs.ucsb.edu echang@ece.ucsb.edu

Abstract

Allowing higher-priority requests to preempt ongoing disk IOs is of particular benefit to delay-sensitive multimedia and real-time systems. In this paper we propose *Semi-preemptible IO*, which divides an IO request into small temporal units of disk commands to enable preemptible disk access. We present main design strategies to allow preemption of each component of a disk access—seek, rotation, and data transfer. We analyze the performance and describe implementation challenges. Our evaluation shows that *Semi-preemptible IO* can substantially reduce IO waiting time with little loss in disk throughput. For example, expected waiting time for disk IOs in a video streaming system is reduced 2.1 times with the throughput loss of less than 6 percent.

1 Introduction

Traditionally, disk IOs have been thought of as non-preemptible operations. Once initiated, they cannot be stopped until completed. Over the years, operating system designers have learned to live with this restriction. However, non-preemptible IOs can be a stumbling block for applications that require short response time. In this paper, we propose methods to make disk IOs semi-preemptible, thus providing the operating system a finer level of control over the disk-drive.

Preemptible disk access is desirable in certain settings. One such domain is that of real-time disk scheduling. Real-time scheduling theoreticians have developed schedulability tests (the test of whether a task set is schedulable such that all deadlines are met) in various settings [9, 10, 11]. In real-time scheduling theory, *blocking*[1], or priority inversion, is defined as the time spent when a higher-priority task is prevented from running due to the non-preemptibility of a low-priority task. Blocking degrades schedulability of real-time tasks and is thus undesirable. Making disk IOs preemptible would reduce blocking and improve the schedulability of real-time disk IOs.

[1]In this paper, we refer to blocking as the *waiting time*.

Another domain where preemptible disk access is essential is that of interactive multimedia such as video, audio, and interactive virtual reality. Because of the large amount of memory required by these media data, they are stored on disks and are retrieved into main memory only when needed. For interactive multimedia applications that require short response time, a disk IO request must be serviced promptly. For example, in an immersive virtual world, the latency tolerance between a head movement and the rendering of the next scene (which may involve a disk IO to retrieve relevant media data) is around 15 milliseconds [2]. Such interactive IOs can be modeled as higher-priority IO requests. However, due to the typically large IO size and the non-preemptible nature of ongoing disk commands, even such higher-priority IO requests can be kept waiting for tens, if not hundreds, of milliseconds before being serviced by the disk.

To reduce the response time for a higher-priority request, its waiting time must be reduced. The *waiting time* for an IO request is the amount of time it must wait, due to the non-preemptibility of the ongoing IO request, before being serviced by the disk. The response time for the higher-priority request is then the sum of its waiting time and service time. The *service time* is the sum of the seek time, rotational delay, and data transfer time for an IO request. (The service time can be reduced by intelligent data placement [27] and scheduling policies [26]. However, our focus is on reducing the waiting time by increasing the preemptibility of disk access.)

In this study, we explore *Semi-preemptible IO* (previously called Virtual IO [5]), an abstraction for disk IO, which provides highly preemptible disk access (average preemptibility of the order of one millisecond) with little loss in disk throughput. *Semi-preemptible IO* breaks the components of an IO job into fine-grained physical disk-commands and enables IO preemption between them. It stances of Façade in the same storage system could cooperate, in order to handle larger workloads that hit common back-end devices. We would also like to explore the implications of providing not only performance SLOs, but also availability/reliability guarantees, by dynamically changing the data layout in a way that is transparent to the client hosts.

Acknowledgements

We thank our shepherd, Eran Gabber, the anonymous reviewers and the members of the HP Labs Storage Systems Department for helpful comments on this work. In particular, we thank John Wilkes for suggestions on the choice of SLOs. We are also grateful to Hernan Laffitte for help with repeated disk array reconfigurations.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administrators. In *Conference on File and Storage Technologies (FAST), (Monterey, CA)*, pages 175–188. USENIX, January 2002.

[3] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, 1998.

[4] S. Blake et al. An architecture for differentiated services, December 1998. IETF RFC 2475.

[5] John L. Bruno, Jose Carlos Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.

[6] G. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-time Systems*, 24(1–2):7–24, July 2002.

[7] N. Christin and J. Liebeherr. The QoSbox: A PC-router for quantitative service differentiation in IP networks. Technical Report CS-2001-28, University of Virginia, November 2001.

[8] IBM Corp. *MVS/DFP V3R3 System Programming Reference*, 1996. SC26-4567-02.

[9] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM Symposium on Communications Architectures and Protocols (Austin, Texas)*, pages 1–12, Sept 1989.

[10] J. Gemmell, H. Vin, D. Kandlur, V. Rangan, and L. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, 28(5):40–49, 1995.

[11] P. Goyal and H. Vin. Generalized guaranteed rate scheduling algorithms: a framework. *IEEE/ACM Transactions on Networking*, 5(4):561–571, 1997.

[12] Hewlett-Packard Company, Palo Alto, CA. *HP SureStore E Disk Array FC60 User's Guide*, 2000. Pub. No. A5277-90001.

[13] Hewlett-Packard Company, Palo Alto, CA. *OpenMail Technical Reference Guide*, 2.0 edition, 2001. Part No. B2280-90064.

[14] Hewlett-Packard Company, Palo Alto, CA. *HP StorageApps sv3000 White Paper*, 2002.

[15] Hewlett-Packard Company, Palo Alto, CA. *HP StorageWorks Virtual Arrays, VA7000 Family, User and Service Guide*, 2002. Pub. No. A6183-96004.

[16] FalconStor Inc. Ipstor: Build an end-to-end IP-based network storage infrastructure. White paper. http://www.falconstor.com, 2001.

[17] T. Madell. *Disk and File Management Tasks on HP-UX*. Prentice Hall, Upper Saddle River, NJ, 1996.

[18] J. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, 1987.

[19] SANSymphony version 5 datasheet. http://www.datacore.com, 2002.

[20] G. Schreck. Making storage organic. Technical report, Forrester Research, May 2002.

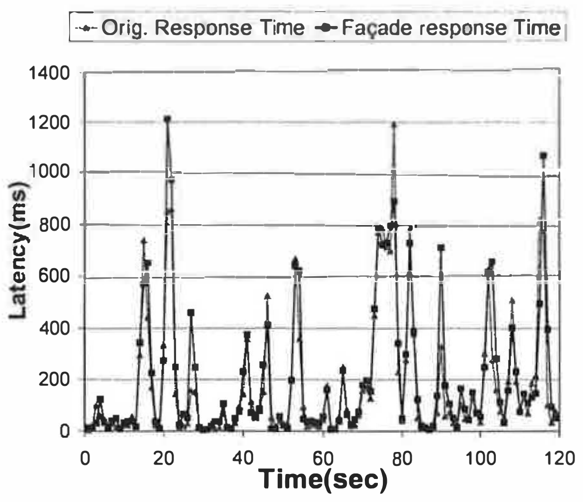
[21] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. *Real-Time Systems*, 22(1-2):9–48, January 2002.

[22] David G. Sullivan and Margo I. Seltzer. Isolation with flexibility: A resource management framework for central servers. In *Proc. of the USENIX Annual Technical Conference (San Diego, California)*, pages 337–350, 2000.

[23] J. Turner. New directions in communications. *IEEE Communications*, 24(10):8–15, October 1986.

[24] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *International Workshop on Quality of Service (Karlsruhe, Germany)*, pages 75–91, June 2001.

[25] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 96–108, Copper Mountain, CO, December 1995. ACM Press.



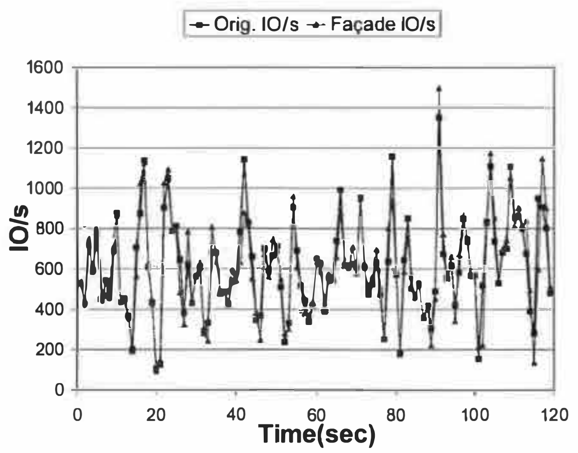
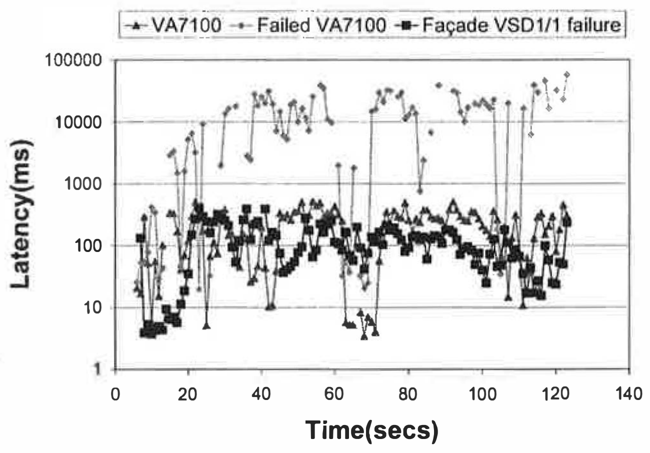


Figure 9. Façade overhead.



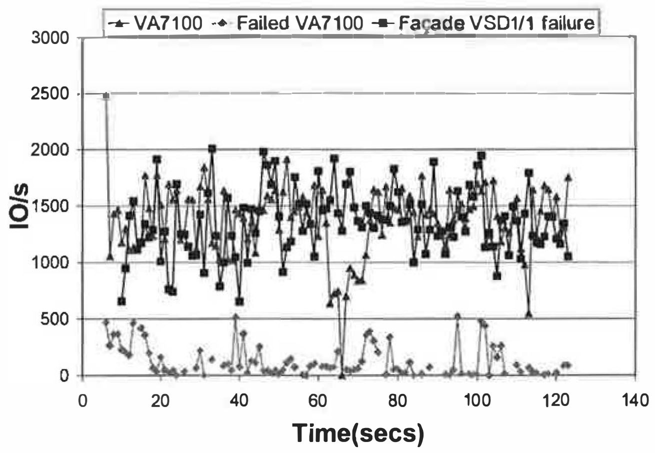
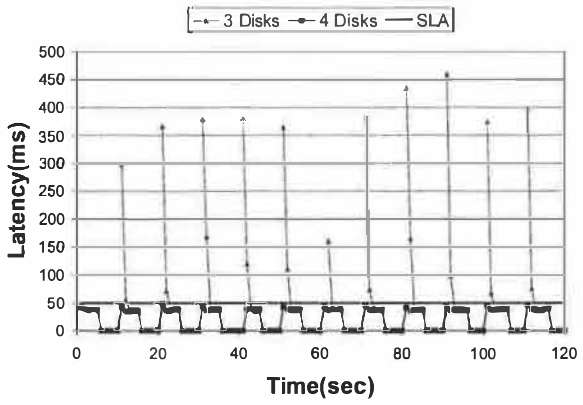


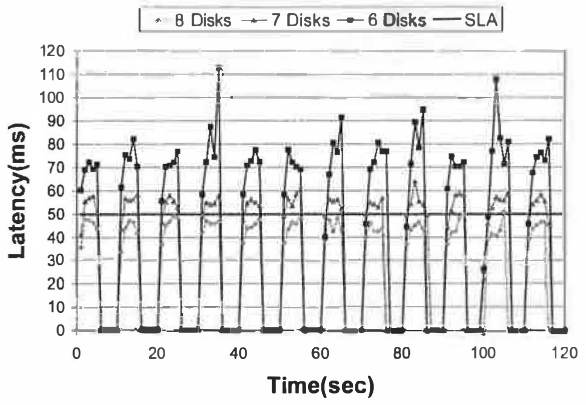
Figure 10. Latency and throughput for Openmail trace on VA7100 LU, VA 7100 LU with one disk failed, and Façade VSD on FC-60 with one disk failed.

commercial disk arrays, show that Façade meets SLOs with a very high probability while making efficient use of the storage resources. Façade can significantly reduce the hardware resources required to support a combination of workloads with tight latency requirements. Façade is capable of supporting workloads with performance requirements that change over time and it can handle as stringent an SLO as the underlying device can support. The performance penalty introduced by the additional processing layer is also negligible, and far outweighed by the resulting advantages. We believe that fully-adaptive storage virtualization appliances such as Façade should play a fundamental role in storage system design; not only does Façade provide fine-grain QoS enforcement, but it also adapts very quickly to changes in the workload, and is not strongly dependent on accurate workload characterizations or storage device models in order to function.

This work has postulated the existence of an admission control component. In future work, we hope to demonstrate the use of Façade to facilitate an adaptive admission control mechanism. Another potential extension for this work is to allow each client application to specify an elasticity coefficient [6] (i.e., a measure of how tolerant it is to deviations from the SLO) and exploit that flexibility in the controller algorithm. Also, multiple in-



With Façade



Without Façade

Figure 8. VSD1 latency with and without Façade: the workload requires 8 disks to meet its target without Façade and only 4 disks with Facade.

load need not bear the entire brunt of the disk failure: instead, the effect can be reduced by spreading the performance loss over several workloads or shifted to workloads that can better tolerate the impact. Additionally, if the throughput of the shared disks is not completely used in normal mode then the impact of the failure can be reduced yet farther. We demonstrate this in the following experiment.

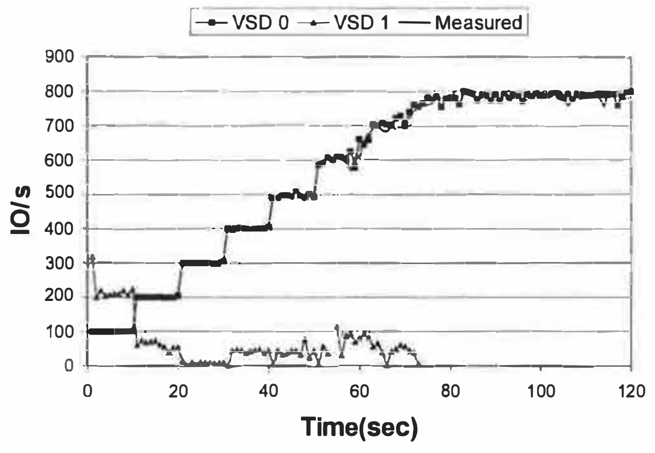
We measured the throughput and latency of replaying an Openmail trace in four configurations: (1) on a VA-7100 LU (without Façade control); (2) on a VA-7100 LU with one disk failed; (3) simultaneously, with small time offsets, on 2 Façade VSDs on an FC-60, each configured with SLOs to match the performance characteristics of a VA-7100 LU; and (4) in the same configuration as (3), but with one disk failed on the FC-60. On the FC-60 configurations, there is an additional background load of 450 IOs/sec with no latency requirement.

Figure 10 shows time plots of latency and throughput for one trace-replay for configurations (1), (2) and (4). The VA-7100 LU plot serves as a baseline for comparison and as the SLO for the Façade plot. For the uncontrolled VA-7100 LU, the latency increases by two orders of magnitude when a disk fails, and the throughput drops by more than one order of magnitude compared with the normal-mode VA-7100 baseline. We have not shown plots for the Façade VSD on the FC-60 in normal mode (configuration 3) to avoid cluttering the figures; however, these would show a performance similar to the normal-mode VA-7100. The plots for the Façade VSD on the FC-60 with one failed disk also show a performance almost identical to the VA-7100 normal mode performance. The effects of the disk failure are entirely absorbed by the background workload (not shown), which suffers a 10% drop in throughput compared with the normal-mode FC-60 operation.

We have demonstrated in Section 4.1 that Façade faithfully matches the specified SLO when the array it is running on is operating normally. This experiment shows further that, if possible, the SLO will be matched also in degraded mode.

5 Conclusions and future work

Traditional solutions to the problem of providing guaranteed performance to independent, competing clients are either inaccurate or substantially more expensive than necessary. By virtue of its fine-grain monitoring and decision-making, Façade is able to use resources much more efficiently, and to balance the load among multiple back-end devices while satisfying the performance requirements of many different client applications. Our experiments, run on a representative mid-range storage system with



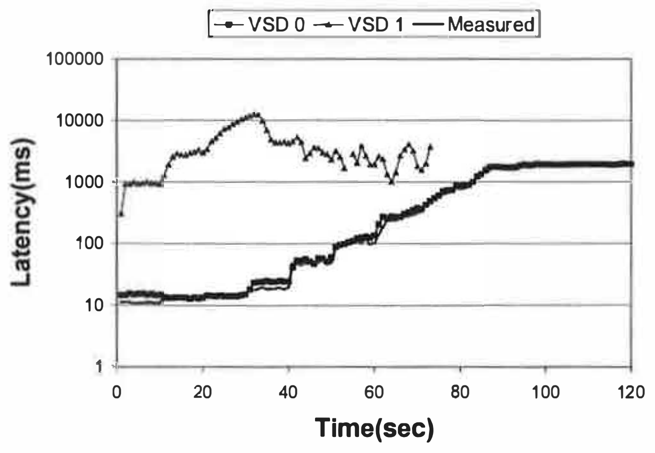


Figure 6. Façade matches maximum (dedicated) array performance. The "measured" data is VSD0 on a dedicated array without Façade, VSD0 and VSD1 are together with Façade control. Measured and VSD0 have complete overlap.

increasing numbers of disks, starting with 3 disks, until the latency target was met, with and without the use of Façade.

Figure 8 shows time plots of latency for VSD1 both with and without Façade. This figure shows that without Façade, 8 disks are required to fully meet the latency targets. With 7 disks the targets are missed by 10-20% and with 6 disks the latencies are almost double the target values.

Using Façade, considerably fewer resources are required to meet the target latencies. With only 3 disks Façade is usually able to meet the latency requirement; however, when a burst of requests starts, there is a latency spike similar to those seen in Section 4.4, due to the time required for draining the device queue. With 4 disks, there are sufficient resources in the system to drain the queue quickly, and the latency spikes disappear. Using Façade, the SLO latency can be met with 50% fewer resources.

4.6 Façade overhead

Since Façade adds an additional control layer between the application and the storage device that modulates the queue depth at the storage device, it is possible that there is some loss of efficiency at the storage device. We measure the Façade overhead in this experiment.

For this experiment, we emulated a VA-7100 LU on itself, as follows. We measured the mean latency for reads and writes for a VA-7100 LU over a range of offered loads, and used this curve as an SLO. We formed a Façade VSD using this SLO and the same LU as the underlying device. We then replayed an Openmail trace against this VSD and (separately) directly against the underlying VA-7100 LU, measuring the throughput and latency over time for both. (A real trace was used for this test because the SLO was formed from measurements based on synthetic workloads.) Since the only difference between the two cases is the use of a Façade layer, the overhead due to Façade should appear as a loss of performance for the VSD run as compared to the run against the VA-7100 LU. Figure 9 shows the time plots of latency and throughput for both cases. It is clear that the performance of the LU under this workload is mostly unaffected by adding the Façade layer: overall, adding the Façade layer left throughput unchanged, and increased the average latency by less than 2%. We conclude that Façade adds a negligible overhead.

4.7 Performance under failure

When workloads are segregated on separate disk-groups in order to provide performance isolation, the impact of a disk failure on a workload can be dramatic. When multiple workloads are consolidated onto a shared group of disks, a single work-

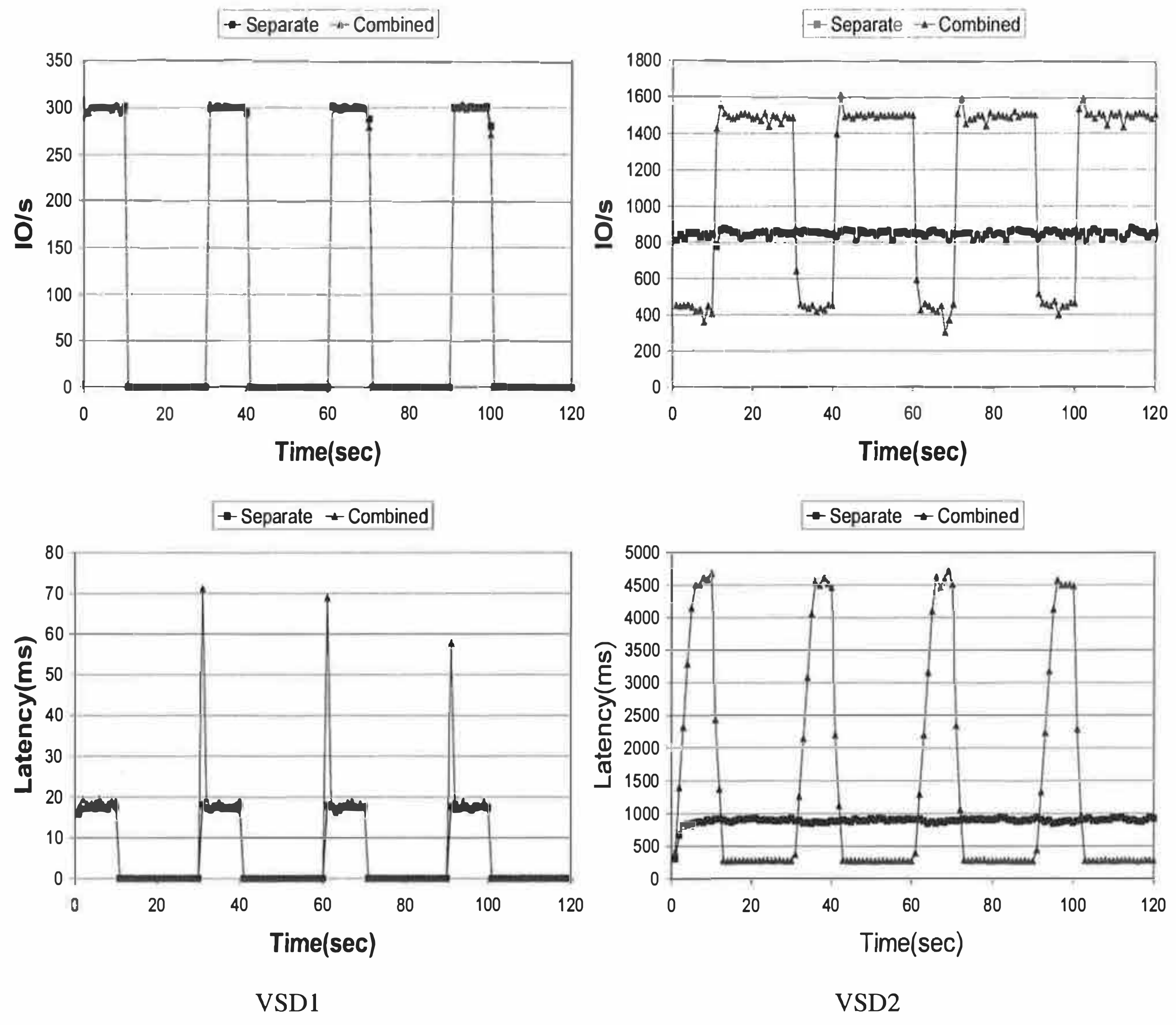
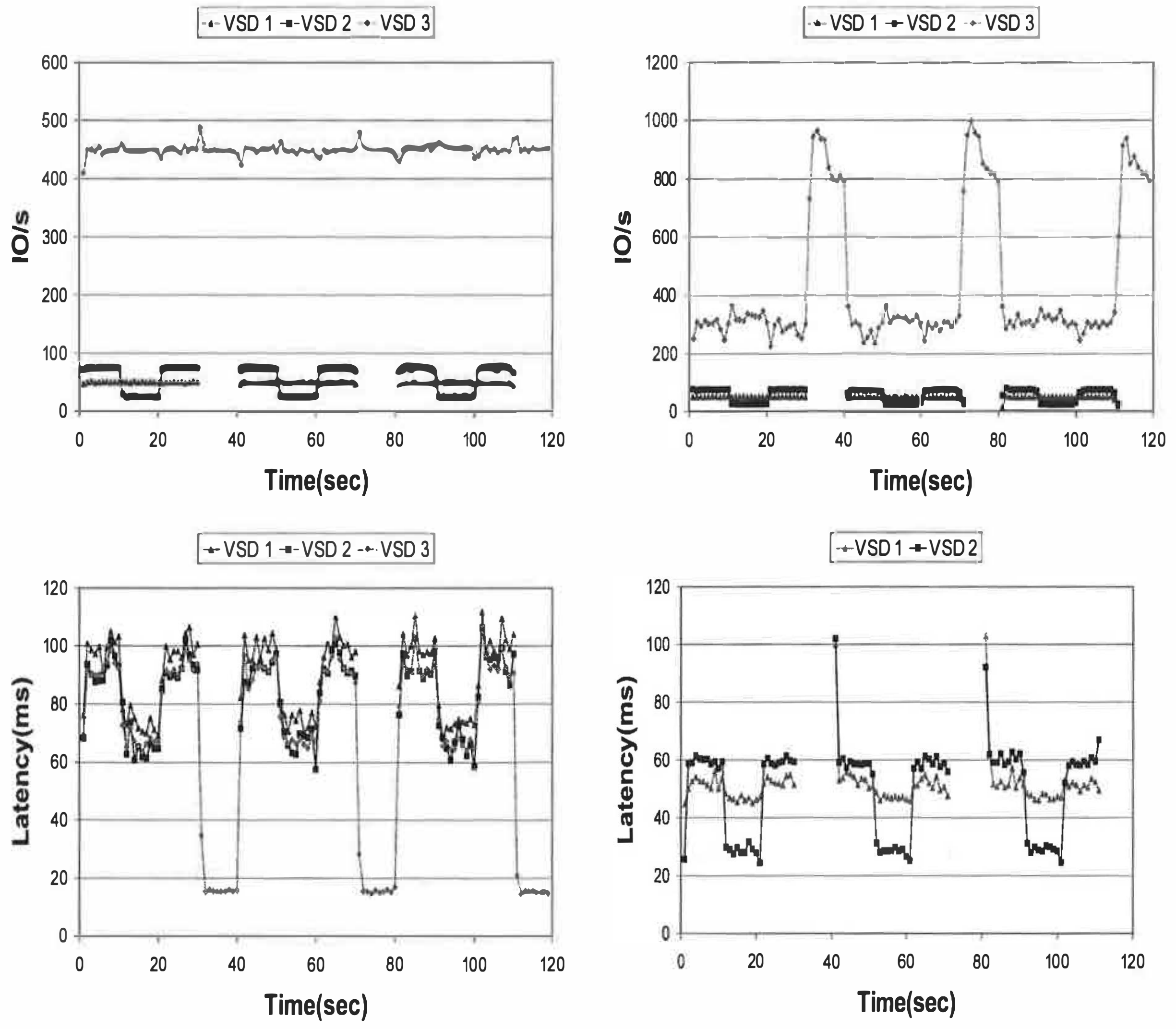


Figure 7. Combining VSDs on single hardware gives higher throughput to VSD2

burst of requests begins. The spike subsides quickly — within one time window. The spike occurs because Façade builds up a large device queue when VSD1 is off, in order to maximize the throughput of VSD2. When VSD1 comes on, the controller quickly reduces the device queue target, but it takes some time for the device queue to drain sufficiently that the VSD1 SLO latency target can be met.

4.5 Resource utilization

To test how efficiently Façade uses resources, we measured the size of the array required to support a workload, with and without Façade. We used the same workloads as in the first performance isolation experiment: VSD1, a synthetic workload of 200 IOs/sec, repeatedly on for 5 seconds and off for 5 seconds, with a latency target of 50ms when on, and VSD2, which requests 500 IOs/sec with a latency bound of 4000ms. We run two workloads together against a series of FC-60 RAID5 LUs with

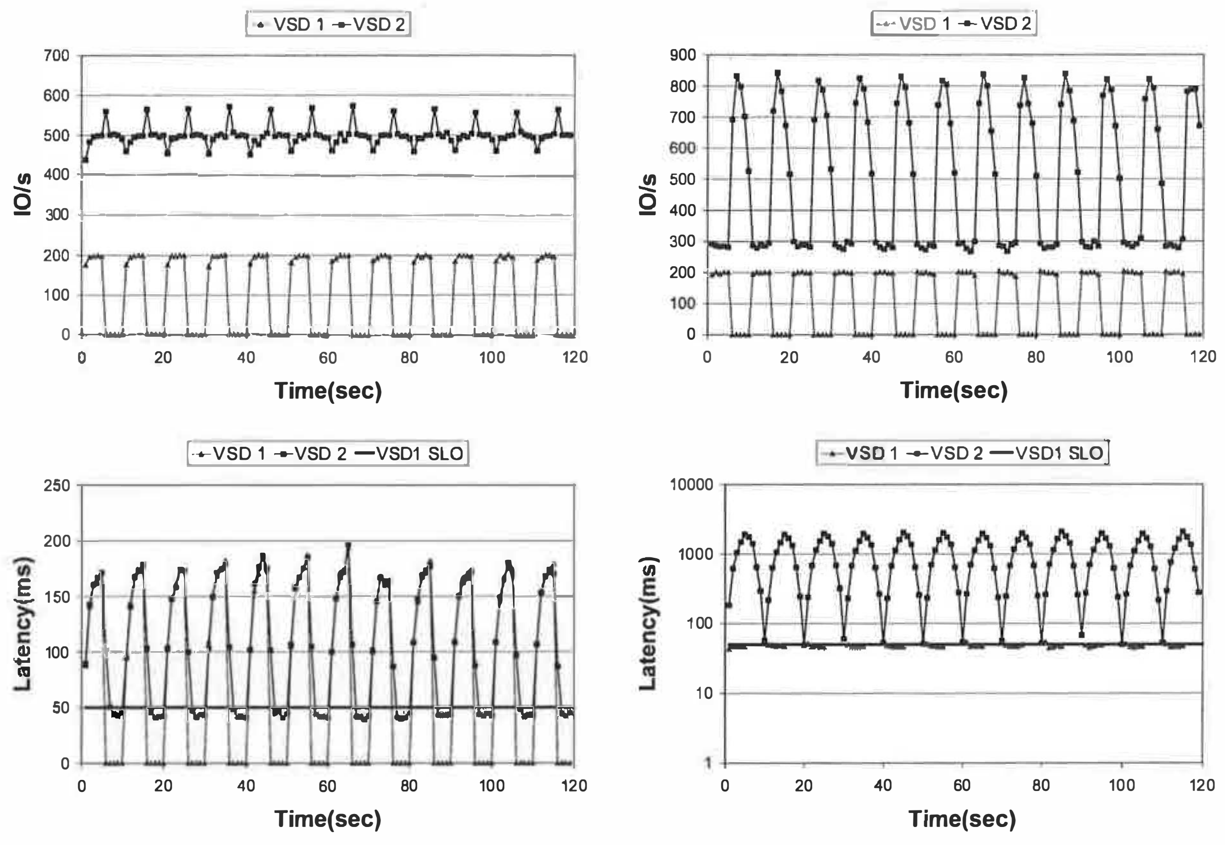


(a) Without Façade control

(b) With Façade control

Figure 5. Performance isolation experiment 2.
Latencies for VSD3 have been dropped from Figure 5(b) for clarity.

10 seconds, and then is off for 20 seconds; when on, it has a latency requirement of 20ms. VSD2 is a greedy workload (representing, for example, a backup); it has no specific latency requirement but will read data as quickly as possible. Figure 7 shows time plots of latency and throughput for VSD1 and VSD2 for two cases: (1) baseline: VSD1 and VSD2 each on a separate FC-60 LU and (2) VSD1 and VSD2 with their data striped across both LUs, using Façade. Our results show that when VSD1 and VSD2 run on separate LUs, VSD2 receives a steady throughput of approximately 847 IOs/sec. When VSD1 and VSD2 use both LUs under Façade, the throughput received by VSD2 is more variable, generally ranging between 450–1500 IOs/sec; however, the average throughput is 1146 IOs/sec, substantially higher than the throughput using separate LUs. VSD1 receives its required throughput and SLO latency in both cases; however, there is a spike in the latency whenever a



(a) Without Façade control

(b) With Façade control

Figure 4. Performance isolation experiment 1.

LU can support for that workload. We use this to test how stringent an SLO Façade can support. We measured the latencies at various IO rates on an otherwise unloaded FC-60 LU using our standard synthetic workload. We used this latency-throughput curve as the SLO for a Façade VSD, VSD0, running on the FC-60 LU; this should be the most stringent SLO the FC-60 LU can support for this workload. The workload for VSD0 has a gradually increasing IO rate. We also added a greedy workload with no specific latency requirement, VSD1, that issues IOs as quickly as possible to add load to the LU. More precisely, VSD1 is a synchronous (closed) workload with 2000 outstanding IOs (4KB random reads) and zero think-time.

Figure 6 shows the measured latency and throughput for VSD0 and VSD1. VSD0 tracks quite faithfully the SLO, which is the performance of the workload on a dedicated array and VSD1 is completely starved out after 80 seconds. This indicates that Façade can support the most stringent SLO that the underlying device can support.

4.4 Multiplexing

While performance isolation can be provided conventionally to workloads by segregating their data on LUs residing on separate hardware components, there can be an overall loss of performance as a result. (Equivalently, it may be necessary to overprovision the storage for each workload in order to meet performance requirements.) We show in this experiment that combining workloads on the same hardware using Façade can provide substantially better performance than segregating the workloads.

There are two workloads: VSD1 is a bursty workload that repeatedly requests 300 IOs/sec for quest rate, but no latency requirement. Figure 3 is a throughput-latency plot showing the measured latency of each throughput range for the Façade VSD and the target based on the SLO. As the graph shows, the Façade VSD latency remains slightly lower than the latency bound in the SLO.

4.2 Performance isolation

An essential property that Façade should satisfy is performance isolation: workloads on different VSDs should not interfere with each other. In particular, the performance of the workload on one Façade VSD should not be fall below its SLO due to bursts in the load on another Façade VSD sharing the same physical hardware. We verify here how well our implementation provides this property.

Figure 4 shows time-plots of latency and throughput for two Façade VSDs, both using same FC-60 LU as a back-end device. VSD1 has a bursty workload: 200 IOs/sec, repeatedly on for 5 seconds and off for 5 seconds. VSD1 also has a relatively tight latency bound of 50ms: the corresponding SLO is (200IOs/sec, 50ms, 50ms)). VSD2 has a stable workload of 500 IOs/sec, but a much looser latency requirement of 4000ms; the corresponding SLO is ((500IOs/sec, 4000ms, 4000ms)). Both workloads are 4KB IOs, 67% reads and 33% writes.

The latency time-plot shows that VSD1 receives the latency it requires—while on, its average latency is approximately 50ms. The latency and throughput seen by VSD2 varies to comply with the requirements of VSD1; however, the requirements of the VSD2 SLO are also met. In particular, the latency provided by VSD2 is affected at approximately the same rate of change when a burst begins (increasing) and when it ends (decreasing). This is because the bursty workload has the more stringent latency requirement, so requests for the other workload are delayed in Façade, and spread evenly over a longer period of time; in fact, some VSD2 requests are delayed almost until the following burst starts. For VSD2, throughput varies between 300 and 800 IOs/sec, and the latency goes as high as 2000ms. When the same workloads are run on the FC-60 LU without Façade control, they are clearly not isolated from one another, as shown in Figure 4(a). The latency for VSD1 exceeds its SLO latency ($50ms$) regularly.

Workload	Without Façade	With Façade
VSD1	75 ms	42 ms
VSD2	72 ms	43 ms
VSD3	71 ms	3922 ms

Table 1. Performance isolation experiment 2: Standard deviation of latency

Figure 5 shows an experiment with a more complex set of workloads. In this case we have three workloads, VSD1, VSD2, and VSD3. VSD1 is a workload running for 30s on, 10s off at 50 IO/s with a latency target of 55ms. VSD2 is a workload runs for 10s at 75 IO/s, 10s at 25 IO/s, 10s at 75 IO/s and then off for 10s. When VSD2 is issuing 25 IO/s its latency target is 30ms when it is issuing 50 IO/s its latency target is 60ms. VSD3 issues requests continuously and has a high latency target of 2000ms. Even with this complex set of workloads, where the SLO latency target of VSD1 is sometimes higher and sometimes lower than that of VSD2, Façade is able to track and meet all latency targets. Without Façade control, the latencies for VSD1, VSD2 and VSD3 are always similar and both VSD1 and VSD2 regularly miss their latency targets.

For this experiment, we also measured the standard deviation of latency, shown in Table 1. The standard deviation of latency for VSD1 and VSD2, the workloads with tight latency requirements, dropped by a factor of $1.67 - -1.78$ when Façade control was used; the standard deviation of the greedy workload VSD3 increased much. This is consistent with what one would expect intuitively: since Façade controls the latencies that the workloads receive, workloads with low latency requirements see latencies close to their requirement, and therefore a lower variance. Workloads with a high latency tolerance see sometimes low and sometimes high latencies, depending on load, and thus a higher variance.

4.3 Maximum SLO

The performance of a workload running on a dedicated LU gives an upper bound on the SLO that to allow for greater throughput. Formally, this is a non-linear feedback controller implemented by the following equations: Suppose the measured average IO latency for workload W_k is $L(W_k)$, the maximum queue depth achieved in the last control period is Q_{max} and the previous target device queue depth is Q_{old}. Then, the new queue depth target Q_{new} is computed as follows:

$$E = \min_k \frac{latencyTarget(W_k)}{L(W_k)}$$

$$Q_{new} = \begin{cases} E \cdot Q_{old} & \text{if } E < 1, \\ (1+\varepsilon)Q_{old} & \text{else if } Q_{max} = Q_{old}, \\ Q_{old} & \text{otherwise.} \end{cases}$$

Here, ε is a small positive value; we use $\varepsilon = 0.1$. The intial queue depth is set to 200 entries.

4 Experimental evaluation

We empirically validated the accuracy, stability and efficiency of our techniques for providing virtual stores with QoS guarantees through experiments on two modern commercial arrays. The FC-60 [12] used in our experiments is a mid-range array with 30 Seagate Cheetah ST118202LC disks (10,000rpm, 18.21GB each), and two controllers with 256MB of NVRAM cache in each. We set up a RAID5 Logical Unit (LU) using 6 disks on this array. The VA-7100 [15] is a small array with 10 Seagate Cheetah ST318451FC disks (15000rpm, 18.35GB each) and two controllers each with 256MB of NVRAM cache. We configured two AutoRAID [25] LUs on this. Each array is connected to an HP 9000-N4000 server through a Brocade Silkworm 2800 switch using two FibreChannel links.

We employed both synthetic and trace workloads in this evaluation, using the Buttress workload generator, which can produce synthetic workloads and replay traces. We used synthetic traces consisting of 67% reads and 33% writes, distributed randomly over the LU; the I/O rates are described with the experiments and the workloads were asynchronous (open) except where specified. The trace workload is an I/O trace from a production OpenMail email server [13]. For each workload execution, we collected traces of I/O activity at the device driver level, including I/O submission and completion times, read/write characteristics, size and logical address information. These traces were then analyzed to provide throughput, latency and other statistics.

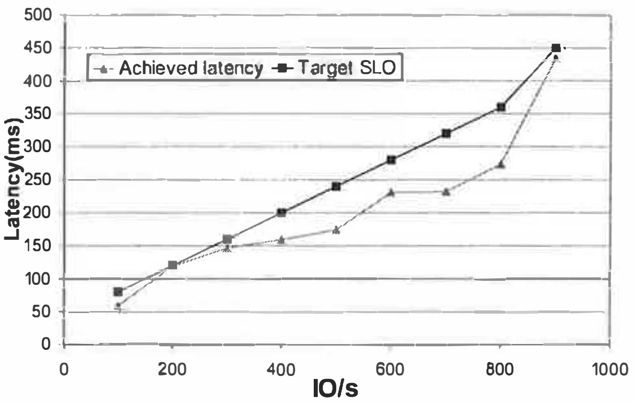


Figure 3. Latency at various IO rates is compared with SLO for a Façade VSD on a FC-60 LU.

We implemented Façade as a software layer between Buttress and the device driver. A Virtual Storage Device (VSD) is a virtual LU provided by the Façade layer. It has an associated Service Level Objective (SLO) and stores its data on a real (physical) LU. One or more VSDs may use the same underlying real LU as a backend device.

We now describe a series of experiments designed to test how well Façade meets its goals: to match the SLO, to provide performance isolation between workloads, to emulate LUs on a different disk array, and to operate without significantly reducing the overall efficiency of the underlying hardware. In each case, the SLO window size is 1 second.

4.1 SLO compliance

This experiment tests how well Façade ensures compliance with a SLO. A Façade VSD was created on an FC-60 LU, with a specified SLO. We then ran a synthetic workload with 67% reads and 33% writes, increasing the IO rate in increments of 50 IOs/sec against the VSD. There is an additional background workload with a high IO re-components that determine when these requests are sent to the underlying storage device: an earliest deadline first (EDF) IO scheduler, a statistics monitor that collects I/O statistics, and a controller that periodically adjusts targets for device queue depth and workload latency. We describe these in order.

IO scheduler

Based on periodic input from the controller, the IO scheduler maintains a target queue depth value and per-workload latency targets, which it tries to meet using Earliest Deadline First (EDF) scheduling. The deadline for a request from workload W_k is $arrivalTime(W_k) + latencyTarget(W_k)$, where $arrivalTime(W_k)$ is its arrival time and $latencyTarget(W_k)$ is a target supplied for W_k by the controller. The deadline for the workload W_k is the deadline of its oldest pending request.

The scheduler polls the device queue depth and the input queues periodically (every 1ms in our implementation) and also upon IO completions. Requests are admitted to into the device queue in two cases. (1) If the device queue depth is now less than the current queue length target (supplied by the controller — see ahead), the scheduler repeatedly picks the workload with the earliest deadline and sends the first request in its queue to the device until the device queue depth target is met. (2) If the deadline for any workload is already past, the past-due requests from that workload are scheduled even if this causes the queue depth target to be exceeded. Since the intent of controlling queue depth is to allow workloads with low latency requirements to satisfy their SLOs, it is not sensible to throttle these workloads: this rule allows newly-arrived low-latency workloads to be served even as the queue depth adapts.

Statistics monitor

The monitor receives IO arrivals and completions. It reports the completions to the IO scheduler, and also computes the average latency and read and write request arrival rates for active workloads every P seconds and reports them to the controller. This control period length P is a tuneable parameter; we used $P = 0.05$.

Controller

The controller periodically (every P seconds) adjusts the target workload latencies and the target device queue length. The target workload latencies must be adjusted because, as the workload request rates vary, Façade must give those requests a different latency based on the workload SLO. The device queue depth must also be adjusted to meet these varying workload requirements. The controller tries to keep the device queue as full as possible while still meeting latency targets, since a full device queue improves device utilization and the throughput it can produce. However, long device queues usually mean long latencies; hence, when any workload demands a low latency, the controller reduces the target queue depth. Reducing the queue size ensures that there are not too many outstanding I/Os in the device queue when an I/O requiring low latency arrives. When it arrives, it will be the next one to be sent to the device, and it will execute faster because the device has fewer outstanding I/Os.

The controller uses the IO statistics it receives from the monitor every P seconds to compute a new latency target based on the SLO for each workload as follows. Formally: suppose the SLO for workload W_k is $((r_1, tr_1, tw_1), (r_2, tr_2, tw_2), \ldots, (r_n, tr_n, tw_n))$ with a window w, and the fraction of reads reported is f_r. Let $r_0 = 0$, $r_{n+1} = \infty$, $tr_{n+1} = tw_{n+1} = \infty$. Then

$$latencyTarget(W_k) = tr_i f_r + tw_i(1 - f_r)$$
$$\text{if } r_{i-1} \leq readRate(W_k) + writeRate(W_k) < r_i.$$

The controller also computes a (possibly) new target queue depth for the storage device, based on the IO latencies measured in the previous control period and the current latency targets. If any workload has a new target latency lower than that measured in the previous control period, the queue depth target is reduced proportionately. On the other hand, if the new target latencies are all larger than those achieved in the previous control period, then the queue depth target can be increased. We check if the queue depth in the previous control period was in fact limited by the queue depth target; if so, we raise the new queue depth target slightly

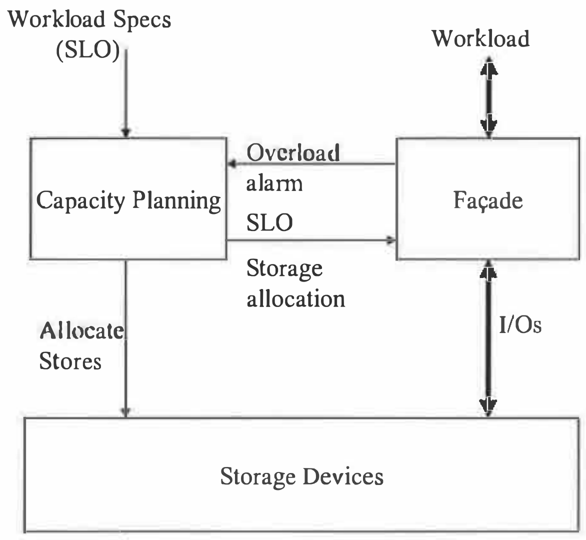


Figure 1. Façade controller in a storage management system.

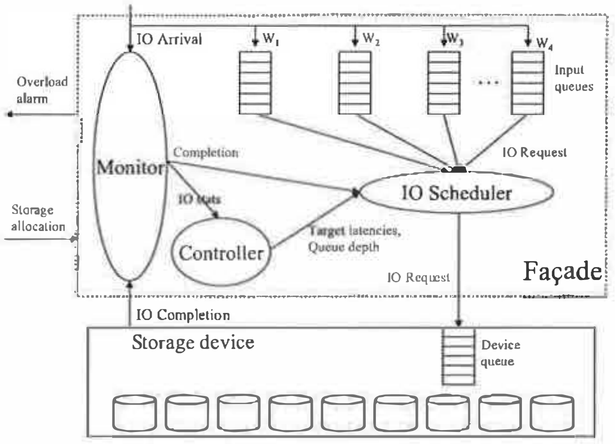


Figure 2. Façade architecture.

formance isolation between dynamically varying workloads.

Figure 1 shows the flow of information through the storage management system. Workload performance requirements, specified as an SLO, are given to the capacity planner (possibly by a system administrator). The capacity planner allocates storage on a storage device, possibly using models to determine if the device has adequate bandwidth to meet the requirements of the workload in addition to those already on it. It passes the allocation information and the SLO requirements to Façade. The workload sends I/O requests to Façade, which communicates with the storage device to complete the requests. If Façade is unable to meet the SLO requirements, this is communicated to the capacity planner, possibly triggering a re-allocation.

A workload SLO consists of a pair of curves specifying read and write latency as a function of the offered request rate, plus a time window length w. Average latency over a window should not exceed the weighted average of the specified latency bounds for the read/write mix and offered request rate during the window. Formally, we represent the two curves discretely as a vector of triples $((r_1,tr_1,tw_1),(r_2,tr_2,tw_2),\ldots,(r_n,tr_n,tw_n))$ where $0 < r_1 < \ldots < r_n$. We divide time into windows (epochs) of length w. For a workload with fraction of reads f_r, the average latency over any time window should not exceed $f_r tr_i + (1-f_r)tw_i$ if the offered request rate over the previous window is less than r_i. This formula implies a latency bound of tr_i for read only workloads, tw_i for write-only workloads, and a linear interpolation between the two bounds for mixed read/write workloads. By default, we assume $r_0 = 0$ and $r_{n+1} = tr_{n+1} = tw_{n+1} = \infty$: in other words, there is no bound on the average latency if the offered load exceeds r_n. For example, consider an application that generates up to 100 transactions per second, each of which causes up to 3 IOs. The transactions are required to complete in 100ms on the average. In this case, one might require an average latency of no more than 33ms per IO so long as the IO rate is no more than 300 IOs/sec, leading to the SLO $((300,0.03,0.03))$.

Façade structure

Façade implements SLOs through a combination of real-time scheduling and feedback-based control of the storage device queue (see Figure 2). This control is based on very simple assumptions about the storage device: we assume only that reducing the length of the device queue reduces the latency at the device, and increasing the device queue may increase throughput. These properties are satisfied by most disks and disk arrays.

Requests arriving at Façade are queued in per-workload input queues. Façade has three main eralize to shared storage systems [24]: dropping SCSI packets to relieve congestion is not an option, no traffic shapers exist within the system, the performance of storage devices is much more dependent on their current states than in the case of network components, and simple linear models are completely inadequate for performance prediction.

Wilkes suggested the notion of a storage QoS controller that can be given per-workload performance targets and enforces these using performance monitoring, feedback based rate control and traffic shaping.

Prior solutions have relied upon real-time scheduling techniques [11]. Work on multimedia servers [10] has primarily addressed the case in which multiple similar streams must be serviced; our version of the problem is more challenging, as we have to process an arbitrary mix of workloads, client hosts do not cooperate with our throttling scheme, and we cannot tolerate long startup delays when a workload arrives into the system. The YFQ disk scheduling algorithm [5] is an approximate version of generalized processor sharing that allows applications to reserve a fixed proportion of a disk's bandwidth. Sullivan used YFQ with lottery scheduling to manage disk resources in a framework that provides proportional share allocation to multiple resources [22]. While YFQ does isolate the performance offered to different applications, it does not support have the notion of per-stream average latency bounds as Façade does. The Cello framework [21] arbitrates disk resources among a heterogeneous mix of clients. It depends on correctly pigeonholing incoming requests into the a limited set of application classes (e.g., "throughput-intensive, best-effort"), on assigning the correct relative priorities to each class, and on being able to compute accurate estimated service times at the device. Façade is fully adaptive, can process a richer variety of clients, and does not depend on the existence of accurate device models.

The concept of storage virtualization, where strings of storage devices are logically coalesced, has existed for over two decades in the MVS mainframe operating system [8]. Modern operating systems include some form of logical volume manager (e.g., HP-UX LVM [17]), which stripe fault-tolerant logical volumes over multiple physical devices. LVMs do not provide performance guarantees and must be configured separately on each client host. A number of current commercial products provide storage virtualization across storage devices in a storage area network, including SANsymphony from DataCore Software [19], IPstore from FalconStor software [16] and the StorageApps sv3000 virtualization appliance [14]; none of them, however, provide performance guarantees for applications.

Automatic system design tools like Minerva [1] and Hippodrome [2] build systems that satisfy declarative, user-specified QoS requirements. They effectively minimize overprovisioning by taking into account workload correlations and detailed device capabilities to design device configuration and data layouts. The whole storage system may be redesigned in every refinement iteration, and there typically is a substantial delay to migrate the data online. As a result, they adapt to changes much more slowly than Façade. We view these tools as complementary to this work: Façade can handle short-term workload variations through adaptive scheduling without migrating data, and possibly postpone the need for a heavyweight system redesign.

3 Structure and components of Façade

Façade is a dynamic storage controller for controlling multiple I/O streams going to a shared storage device, and to ensure that each of the I/O streams receives a performance specified by its service-level-objective (SLO). Façade is designed for use in a storage management system (see Figure 1) with a separate capacity planning component. The capacity planner allocates storage for each workload on the storage device and ensures that the device has adequate capacity and bandwidth to meet the aggregate demands of the workloads assigned to it. This allocation may be changed periodically to meet changing workload requirements and device configurations [2]; however, such reallocation occurs on a time scale of hours to weeks. Façade manages device resources in time scale of milliseconds, to enable SLO compliant per-one SSP to another: for example, "My application runs well on an XP-1024 array; I want performance similar to this."

Traditional techniques for providing guarantees in the networking domain do not readily apply to storage, and adaptation at the application level is extremely rare [24]. The storage system is therefore left with the task of apportioning its resources with very little knowledge of the highly-variable workloads, and subject to constraints implied by protocols originally designed to provide best-effort service. One approach commonly followed to ensure performance isolation is to overprovision to the point that performance is no longer an issue. The resulting system can easily be twice as expensive as a correctly-designed system [1, 20]—a substantial difference for hardware and management costs in the order of millions of dollars. Another alternative is to assign separate physical resources (e.g., separate arrays) to different customers. This inflexible solution is impractical for a heterogeneous system that contains a time-evolving mix of newer and older devices, and makes it difficult to add capacity in arbitrary increments without extensive reconfiguration when customer requirements change. It can also cause overprovisioning by constraining the SSP to the design points of devices in the market: a given customer's workload may use a large fraction of a disk array's bandwidth, but only a small portion of its capacity. Besides, it does not solve the problem of resource allocation: the actual performance characteristics of real arrays are notoriously hard to model and predict.

We propose to solve this problem by adding one level of virtualization between hosts and storage devices. Façade provides the abstraction of virtual stores with performance guarantees, by intercepting every I/O request from hosts and redirecting them to the back-end devices. Unlike raw disk arrays (which do not guarantee predictable performance) Façade provides, by design, statistical guarantees for any client SLO. Virtual stores are not restricted by the design decisions embodied in any particular array, and can effectively shield client hosts from the effects of adding capacity or reconfiguring the back-end.

To evaluate this idea, we implemented a Façade prototype as a software layer between the workload-generator/trace-replayer and the storage device. Façade can also be implemented as a *shim box*, a thin controller that sits between the hosts and the storage. We subjected the prototype to a variety of workloads using several different commercial disk arrays s back-end devices. We then applied Façade's basic mechanism to the particular case of disk array virtualization: providing virtual stores that have the same performance characteristics as existing disk arrays. We found that Façade is able to satisfy stringent SLOs even in the presence of dynamic, bursty workloads, with a minimal impact on the efficiency of the storage device. The performance of Façade virtual storage devices is close to that of the device being emulated, even with failures on the host storage device.

Our experiments indicate that Façade can support a set of workloads with SLOs on a storage device if the device is capable of supporting them. We assume that there is an external capacity planning/admission control component which limits the workloads presented to Façade to what can be supported on the device. Façade can facilitate admission control by indicating when the device is approaching its limits; however, the design of an admission control component is outside the scope of this paper.

The remainder of this paper is as follows. Section 2 describes some related work. We introduce the architecture and internal policies of Façade in Section 3. We then evaluate Façade in Section 4, both on general SLOs and for array virtualization; we then conclude in Section 5.

2 Related work

There is an extensive body of work on networking SLAs [3] and network flow-control methods going back to the leaky bucket throttling algorithm [23] and fair queueing [18, 9]. There are several architectures for providing different levels of services to different flows, including the Differentiated Services (DiffServ) architecture [4] and the QoSBox [7], by mapping groups of flows with similar requirements into a few classes of traffic. Unfortunately, most of the techniques used do not gen-

Façade: virtual storage devices with performance guarantees

Christopher R. Lumb* Arif Merchant† Guillermo A. Alvarez‡
Hewlett-Packard Laboratories

Abstract

High-end storage systems, such as those in large data centers, must service multiple independent workloads. Workloads often require predictable quality of service, despite the fact that they have to compete with other rapidly-changing workloads for access to common storage resources. We present a novel approach to providing performance guarantees in this highly-volatile scenario, in an efficient and cost-effective way. Façade, a virtual store controller, sits between hosts and storage devices in the network, and throttles individual I/O requests from multiple clients so that devices do not saturate. We implemented a prototype, and evaluated it using real workloads on an enterprise storage system. We also instantiated it to the particular case of emulating commercial disk arrays. Our results show that Façade satisfies performance objectives while making efficient use of the storage resources—even in the presence of of failures and bursty workloads with stringent performance requirements.

1 Introduction

Driven by rapidly increasing requirements, storage systems are getting larger and more complex than ever before. The availability of fast, switched storage fabrics and large disk arrays has enabled the creation of data centers comprising tens of large arrays with thousands of logical volumes and file systems, for total capacities of hundreds of terabytes and aggregate transfer rates of tens to hundreds of GB/s.

Such a consolidated data center typically serves the storage needs of the organization it belongs to, or even of multiple organizations. For example, companies that outsource their data storage and management contract the services of a Storage Service Provider (SSP); large organizations may also follow this model internally, to satisfy the requirements of separate divisions. The SSP allocates storage on its own disk arrays and makes it available to the customer over a network. Because the SSP serves multiple customers, it can do so more efficiently than a single customer—who may lack the space, time, money and expertise to build and maintain its own storage infrastructure. But this strength of the SSP can also be its bane: independent workloads compete for storage resources such as cache space, disk, arm, bus, and network bandwidth, and controller cycles.

A customer's contract with an SSP frequently includes a Service Level Agreement that combines a *Service Level Objective* (SLO) specifying the capacity, availability and performance requirements that the provided storage will meet, plus the financial incentives and penalties for meeting or failing to meet the SLO. We concentrate on the problem of providing performance guarantees. A primary requirement is *performance isolation:* the performance experienced by the workload from a given customer must not suffer because of variations on the workloads from other customers. Additional requirements may be imposed by customers moving from internally-managed storage to an SSP, or from

*Current address: Carnegie Mellon University, Hamerschlag Hall, Pittsburgh, PA 15213, USA, valheru@ece.cmu.edu

†Current address: 1501 Page Mill Rd., MS 1134, HP Laboratories, Palo Alto, CA 94304, USA. arif@hpl.hp.com

‡Current address: IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA., alvarezg@almaden.ibm.com

2. We use the legally correct term "SPC1 like", since, as per the benchmark specification, to use the term "SPC1" we must also quote other performance numbers that are not of interest here.

3. Due to extremely large overhead and numerical instabilities, we were not able to simulate LRFU and LRU-2 for larger traces such as ConCat and Merge(P) and for larger cache sizes beyond 512 MBytes.

ACKNOWLEDGMENT

We are grateful to Jai Menon for holding out the contrarian belief that cache replacement policies can be improved. The second author is grateful to his managers, Moidin Mohiuddin and Bill Tetzlaff, for their constant support and encouragement during this work. We are grateful to Brent Beardsley, Pawan Goyal, Robert Morris, William Tetzlaff, Renu Tewari, and Honesty Young for valuable discussions and suggestions. We are grateful to Bruce McNutt and Renu Tewari for the SPC1 trace, to Windsor Hsu for traces P1 through P14, to Ruth Azevedo for the trace DS1, to Gerhard Weikum for the CODASYL trace, to Ken Bates and Bruce McNutt for traces S1-S3, to Song Jiang for sharing his LIRS simulator, and to Sang Lyul Min and Donghee Lee for sharing their LRFU simulator and for various numbers reported in Table IV. We would like to thank Peter Chen, our FAST shepherd, and anonymous referees for valuable suggestions that greatly improved this paper.

REFERENCES

[1] J. Z. Teng and R. A. Gumaer, "Managing IBM database 2 buffers to maximize performance," *IBM Sys. J.*, vol. 23, no. 2, pp. 211–218, 1984.

[2] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proc. USENIX Symp. Internet Technologies and Systems, Monterey, CA*, 1997.

[3] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou, "A middleware system which intelligently caches query results," in *Middleware 2000*, vol. LNCS 1795, pp. 24–44, 2000.

[4] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache performance of the SPEC benchmark suite," Tech. Rep. CS-TR-1991-1049, University of California, Berkeley, 1991.

[5] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 134–154, 1988.

[6] A. J. Smith, "Disk cache-miss ratio analysis and design considerations," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 161–203, 1985.

[7] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, 1994.

[8] M. J. Bach, *The Design of the UNIX Operating System.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

[9] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Comm. ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[10] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Comm. ACM*, vol. 28, no. 2, pp. 202–208, 1985.

[11] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inform. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[12] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Sys. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Sys. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[14] P. J. Denning, "Working sets past and present," *IEEE Trans. Software Engineeing*, vol. SE-6, no. 1, pp. 64–84, 1980.

[15] W. R. Carr and J. L. Hennessy, "WSClock – a simple and effective algorithm for virtual memory management," in *Proc. Eighth Symp. Operating System Principles*, pp. 87–95, 1981.

[16] J. E. G. Coffman and P. J. Denning, *Operating Systems Theory.* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[17] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *J. ACM*, vol. 18, no. 1, pp. 80–93, 1971.

[18] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. ACM SIGMOD Conf.*, pp. 297–306, 1993.

[19] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "An optimality proof of the LRU-K page replacement algorithm," *J. ACM*, vol. 46, no. 1, pp. 92–112, 1999.

[20] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. VLDB Conf.*, pp. 297–306, 1994.

[21] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS Conf.*, 2002.

[22] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–142, 1990.

[23] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Computers*, vol. 50, no. 12, pp. 1352–1360, 2001.

[24] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies," in *Proc. ACM SIGMETRICS Conf.*, pp. 134–143, 1999.

[25] Y. Zhou and J. F. Philbin, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annual Tech. Conf. (USENIX 2001), Boston, MA*, pp. 91–104, June 2001.

[26] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari, "Adaptive caching by refetching," in *NIPS*, 2002.

[27] I. Ari, A. Amer, R. Gramarcy, E. Miller, S. Brandt, and D. Long, "ACME: Adaptive caching using multiple experts," in *Proceedings of the Workshop on Distributed Data and Structures (WDAS), Carleton Scientific*, 2002.

[28] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proc. USENIX Annual Tech. Conf. (USENIX 2002), Monterey, CA*, pp. 161–175, June 2002.

[29] W. W. Hsu, A. J. Smith, and H. C. Young, "The automatic improvement of locality in storage systems." Tech. Rep., Computer Science Division, Univ. California, Berkeley, Nov. 2001.

[30] W. W. Hsu, A. J. Smith, and H. C. Young, "Characteristics of I/O traffic in personal computer and server workloads." Tech. Rep., Computer Science Division, Univ. California, Berkeley, July 2001.

[31] B. McNutt and S. A. Johnson, "A standard test of I/O cache," in *Proc. the Computer Measurements Group's 2001 International Conference*, 2001.

[32] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.

[33] B. McNutt, *The Fractal Structure of Data Reference: Applications to the Memory Hierarchy.* Boston, MA: Kluwer Academic Publishers, 2000.

very small value of the parameter p, that is, it favors frequency over the entire workload. In this case, arguably, there exists a small range of optimal parameters. Now, due to its constant adaptation, ARC will never lock into a single, fixed parameter, but rather keeps fluctuating around the optimal choice. This fluctuations cost ARC slightly over offline optimal FRC_p–in terms of the hit ratio. Hence, for stable workloads, we expect ARC to be slightly worse than offline optimal FRC_p. Nonetheless, the latter is a theoretical construct that is not available in practice. Moreover, even for stable workloads, the value of best fixed, offline parameter p depends on the workload and the cache size. The policy ARC provides a reasonable, online approximation to offline optimal FRC_p without requiring any *a priori* workload-specific or cache size-specific tuning.

H. A Closer Examination of Adaptation in ARC

We now study the adaptation parameter p more closely. For the trace P4 and for cache size $c = 32768$ pages, in Figure 7, we plot the parameter p versus the virtual time (or the number of requests). When p is close to zero, ARC can be thought of as emphasizing the contents of the list L_2, and, when p is close to the cache size, 32768, ARC can be thought of as emphasizing the contents of the list L_1. It can be seen that the parameter p keeps fluctuating between these two extremes. Also, it can be seen that it touches both the extremes. As a result, ARC continually adapts and reconfigures itself. Quite dramatically, the policy ARC can fluctuate from frequency to recency and then back all within a single workload. Moreover, such fluctuations may occur as many times as dictated by the nature of the workload without any *a priori* knowledge or offline tuning. Poetically, at any time, p dances to the tune being played by the workload. It is this continuous, unrelenting adaptation in response to the changing and evolving workload that allows ARC to outperform LRU.

VI. Conclusions

We have reviewed various recent cache replacement algorithms, namely, LRU-2, 2Q, LRFU, LIRS. Performance of these algorithms depends crucially on their respective tunable parameters. Hence, no single universal rule of thumb can be used to *a priori* select the tunables of these algorithms. In addition, we have demonstrated that the computational overhead of LRU-2 and LRFU makes them practically less attractive.

We have presented a new cache replacement policy ARC that is online and self-tuning. We have empirically demonstrated that ARC adapts to a given workload dynamically. We have empirically demonstrated that ARC performs as well as (and sometimes even better than) FRC_p with the best offline choice of the parameter

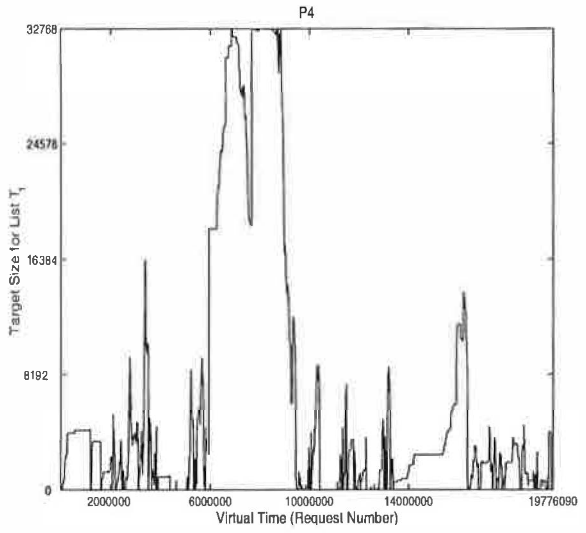


Fig. 7. A plot of the adaptation parameter p (the target size for list T_1) versus the virtual time for the trace P4. The cache size was 32768 pages. The page size was 512 bytes.

p for each workload and cache size. Similarly, we have also shown that ARC which is online performs as well as LRU-2, 2Q, LRFU, and LIRS–even when these algorithms use the best offline values for their respective tuning parameters. We have demonstrated that ARC outperforms 2Q when the latter is forced to be online and use "reasonable" values of its tunable parameters. We have demonstrated that performance of ARC is comparable to and often better than MQ even in the specific domain for which the latter was designed. Moreover, in contrast to MQ, we have shown that ARC is robust for a wider range of workloads and has less overhead. We have demonstrated that ARC has overhead comparable to that of LRU, and, hence, is a low overhead policy. We have argued that ARC is scan-resistant. Finally, and most importantly, we have shown that ARC substantially outperforms LRU virtually uniformly across numerous different workloads and at various different cache sizes.

Our results show that there is considerable room for performance improvement in modern caches by using adaptation in cache replacement policy. We hope that ARC will be seriously considered by cache designers as a suitable alternative to LRU.

Endnotes

1. Our use of *universal* is motivated by a similar use in data compression [11] where a coding scheme is termed universal if–even without any knowledge of the source that generated the string–it asymptotically compresses a given string as well as a coding scheme that knows the statistics of the generating source.

SPC1 like

c	LRU	MQ	2Q	ARC
	ONLINE			
65536	0.37	0.37	0.66	0.82
131072	0.78	0.77	1.31	1.62
262144	1.63	1.65	2.59	3.23
524288	3.66	3.72	6.34	7.56
1048576	9.19	14.96	17.88	20.00

Merge(S)

c	LRU	MQ	2Q	ARC
	ONLINE			
16384	0.20	0.20	0.73	1.04
32768	0.40	0.40	1.47	2.08
65536	0.79	0.79	2.85	4.07
131072	1.59	1.59	5.52	7.78
262144	3.23	4.04	10.36	14.30
524288	8.06	14.89	18.89	24.34
1048576	27.62	40.13	35.39	40.44
1572864	50.86	56.49	53.19	57.19
2097152	68.68	70.45	67.36	71.41
4194304	87.30	87.29	86.22	87.26

TABLE VI. A comparison of hit ratios of LRU, MQ, 2Q, and ARC on the traces SPC1 like and Merge(S). All hit ratios are reported in percentages. The algorithm 2Q is forced to use "reasonable" values of its tunable parameters, specifically, $Kin = 0.3c$ and $Kout = 0.5c$. The page size is 4 KBytes for both traces. The largest cache simulated for SPC1 like was 4 GBytes and that for Merge(S) was 16 GBytes.

P4

c	LRU	MQ	ARC
	ONLINE		
1024	2.68	2.67	2.69
2048	2.97	2.96	2.98
4096	3.32	3.31	3.50
8192	3.65	3.65	4.17
16384	4.07	4.08	5.77
32768	5.24	5.21	11.24
65536	10.76	12.24	18.53
131072	21.43	24.54	27.42
262144	37.28	38.97	40.18
524288	48.19	49.65	53.37

TABLE VII. A comparison of hit ratios of LRU, MQ, and ARC on the trace P4. All hit ratios are reported as percentages. It can be seen that ARC outperforms the other two algorithms.

F. ARC *and* LRU

We now focus on LRU which is the single most widely used cache replacement policy. For all the traces listed in Section V-A, we now plot the hit ratio of ARC versus LRU in Figures 5 and 6. The traces P4, P8, P12, SPC1 like, and Merge(S) are not plotted since they have been discussed in various tables. The traces S1 and S2 are not plotted since their results are virtually identical to S3 and Merge(S). The traces P11, P13, and P14 are not plotted for space consideration; for these traces, ARC was uniformly better than LRU. It can be clearly seen that ARC substantially outperforms LRU on virtually all the traces and for all cache sizes. In

Workload	c	space MB	LRU	ARC	FRC OFFLINE
P1	32768	16	16.55	28.26	29.39
P2	32768	16	18.47	27.38	27.61
P3	32768	16	3.57	17.12	17.60
P4	32768	16	5.24	11.24	9.11
P5	32768	16	6.73	14.27	14.29
P6	32768	16	4.24	23.84	22.62
P7	32768	16	3.45	13.77	14.01
P8	32768	16	17.18	27.51	28.92
P9	32768	16	8.28	19.73	20.28
P10	32768	16	2.48	9.46	9.63
P11	32768	16	20.92	26.48	26.57
P12	32768	16	8.93	15.94	15.97
P13	32768	16	7.83	16.60	16.81
P14	32768	16	15.73	20.52	20.55
ConCat	32768	16	14.38	21.67	21.63
Merge(P)	262144	128	38.05	39.91	39.40
DS1	2097152	1024	11.65	22.52	18.72
SPC1	1048576	4096	9.19	20.00	20.11
S1	524288	2048	23.71	33.43	34.00
S2	524288	2048	25.91	40.68	40.57
S3	524288	2048	25.26	40.44	40.29
Merge(S)	1048576	4096	27.62	40.44	40.18

TABLE VIII. At-a-glance comparison of hit ratios of LRU and ARC for various workloads. All hit ratios are reported in percentages. It can be seen that ARC outperforms LRU–sometimes quite dramatically. Also, ARC which is online performs very close to FRC with the best fixed, offline choice of the parameter p.

Table VIII, we present an at-a-glance comparison of ARC with LRU for all the traces–where for each trace we selected a practically relevant cache size. The trace SPC1 contains long sequential scans interspersed with random requests. It can be seen that even for this trace ARC, due to its scan-resistance, continues to outperform LRU.

G. ARC *is Self-Tuning and Empirically Universal*

We now present the most surprising and intriguing of our results. In Table VIII, *it can be seen that* ARC*, which tunes itself, performs as well as (and sometimes even better than) the policy* FRC_p *with the best fixed, offline selection of the parameter* p. This result holds for all the traces. In this sense, ARC is empirically universal.

It can be seen in Table VIII that ARC can sometimes outperform offline optimal FRC_p. This happens, for example, for the trace P4. For this trace, Figure 7 shows that the parameter p fluctuates over its entire range. Since ARC is adaptive, it tracks the variation in the workload by dynamically tuning p. In contrast, FRC_p must use a single, fixed choice of p throughout the entire workload; the offline optimal value was $p = 0.1c$. Hence, the performance benefit of ARC.

On the other hand, ARC can also be slightly worse than offline optimal FRC_p. This happens, for example, for the trace P8. Throughout this trace, ARC maintains a

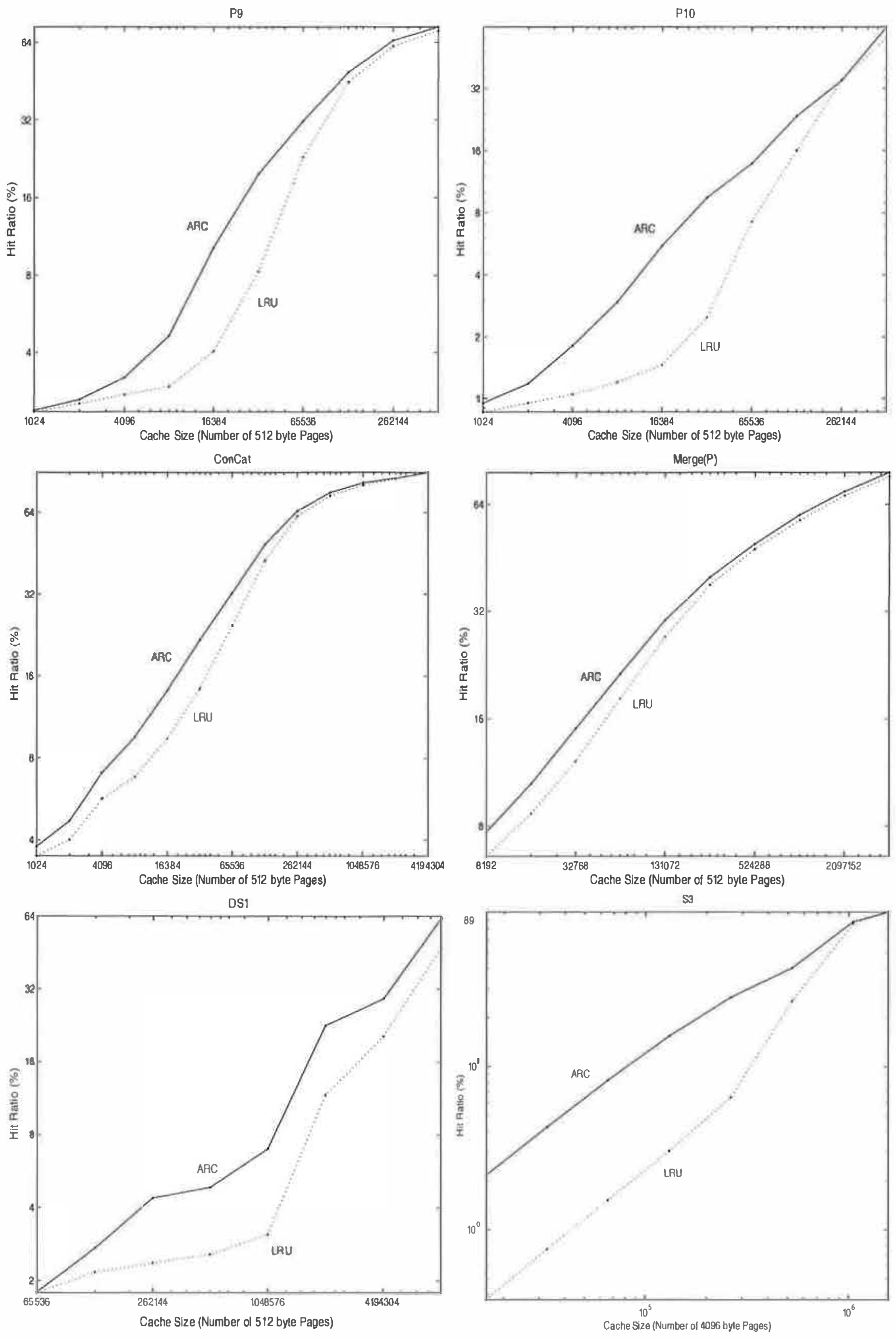


Fig. 6. A plot of hit ratios (in percentages) achieved by ARC and LRU. Both the x- and y-axes use logarithmic scale.

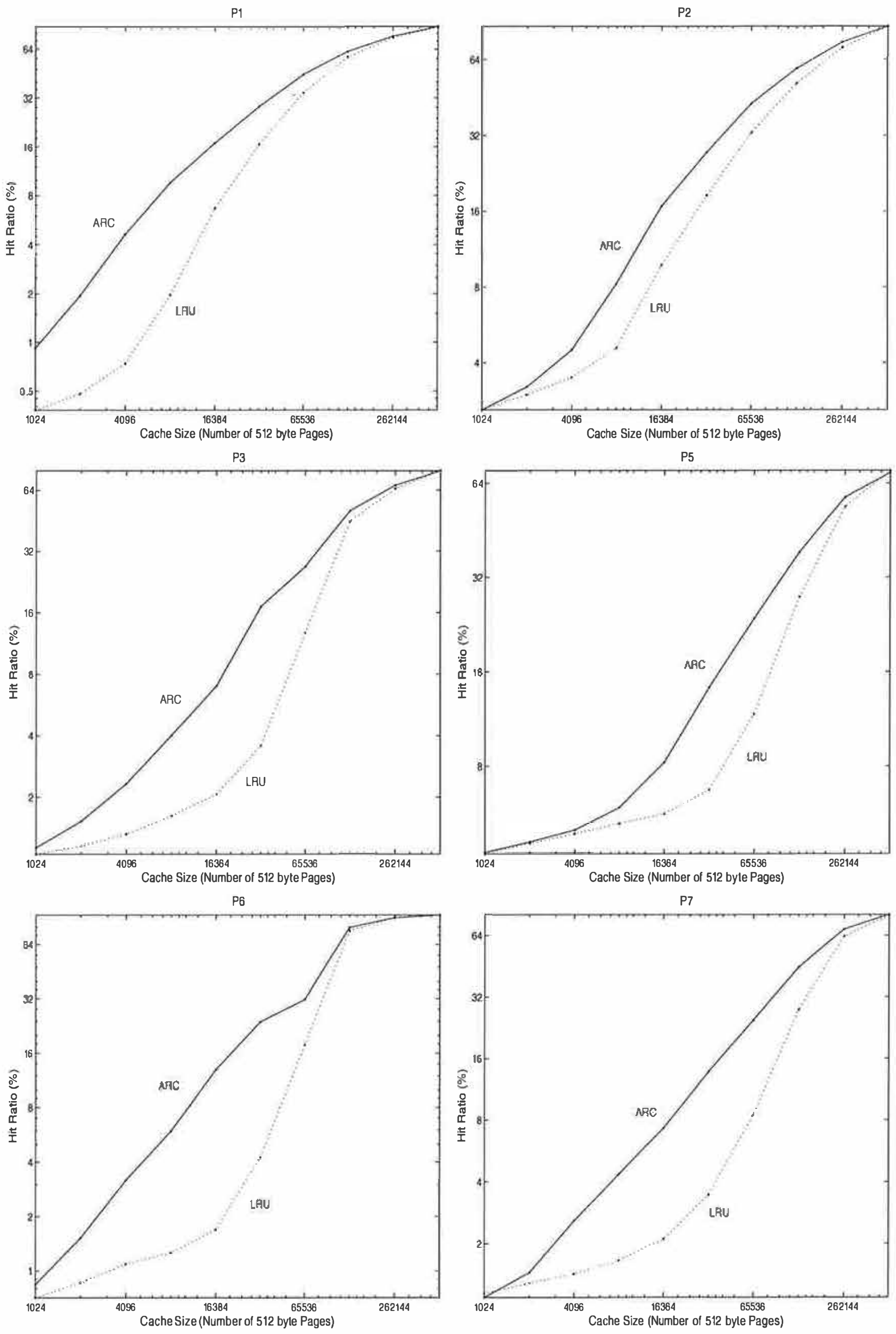


Fig. 5. A plot of hit ratios (in percentages) achieved by ARC and LRU. Both the x- and y-axes use logarithmic scale.

OLTP

c	LRU	ARC	FBR	LFU	LIRS	MQ	LRU-2	2Q	LRFU	MIN
	ONLINE						OFFLINE			
1000	32.83	38.93	36.96	27.98	34.80	37.86	39.30	40.48	40.52	53.61
2000	42.47	46.08	43.98	35.21	42.51	44.10	45.82	46.53	46.11	60.40
5000	53.65	55.25	53.53	44.76	47.14	54.39	54.78	55.70	56.73	68.27
10000	60.70	61.87	62.32	52.15	60.35	61.08	62.42	62.58	63.54	73.02
15000	64.63	65.40	65.66	56.22	63.99	64.81	65.22	65.82	67.06	75.13

TABLE IV. A comparison of ARC hit ratios with those of various cache algorithms on the OLTP trace. All hit ratios are reported as percentages. It can be seen that ARC outperforms LRU, LFU, FBR, LIRS, and MQ and performs as well as LRU-2, 2Q, and LRFU even when these algorithms use the best offline parameters.

P8

c	LRU	MQ	ARC	2Q	LRU-2	LRFU	LIRS
	ONLINE			OFFLINE			
1024	0.35	0.35	1.22	0.94	1.63	0.69	0.79
2048	0.45	0.45	2.43	2.27	3.01	2.18	1.71
4096	0.73	0.81	5.28	5.13	5.50	3.53	3.60
8192	2.30	2.82	9.19	10.27	9.87	7.58	7.67
16384	7.37	9.44	16.48	18.78	17.18	14.83	15.26
32768	17.18	25.75	27.51	31.33	28.86	28.37	27.29
65536	36.10	48.26	43.42	47.61	45.77	46.72	45.36
131072	62.10	69.70	66.35	69.45	67.56	66.60	69.65
262144	89.26	89.67	89.28	88.92	89.59	90.32	89.78
524288	96.77	96.83	97.30	96.16	97.22	97.38	97.21

P12

c	LRU	MQ	ARC	2Q	LRU-2	LRFU	LIRS
	ONLINE			OFFLINE			
1024	4.09	4.08	4.16	4.13	4.07	4.09	4.08
2048	4.84	4.83	4.89	4.89	4.83	4.84	4.83
4096	5.61	5.61	5.76	5.76	5.81	5.61	5.61
8192	6.22	6.23	7.14	7.52	7.54	7.29	6.61
16384	7.09	7.11	10.12	11.05	10.67	11.01	9.29
32768	8.93	9.56	15.94	16.89	16.36	16.35	15.15
65536	14.43	20.82	26.09	27.46	25.79	25.35	25.65
131072	29.21	35.76	38.68	41.09	39.58	39.78	40.37
262144	49.11	51.56	53.47	53.31	53.43	54.56	53.65
524288	60.91	61.35	63.56	61.64	63.15	63.13	63.89

TABLE V. A comparison of ARC hit ratios with those of various cache algorithms on the traces P8 and P12. All hit ratios are reported in percentages. It can be seen that ARC outperforms LRU and performs close to 2Q, LRU-2, LRFU, and LIRS even when these algorithms use the best offline parameters. On the trace P8, ARC outperforms MQ for some cache sizes, while MQ outperforms ARC for some cache sizes. On the trace P12, ARC uniformly outperforms MQ.

E. ARC *and* MQ

It can be seen from Table V that, for the trace P8, ARC outperforms MQ for some cache sizes, while MQ outperforms ARC for some cache sizes. Furthermore, it can be seen that ARC uniformly outperforms MQ, for the trace P12.

The workloads SPC1 and Merge(S) both represent requests to a storage controller. In Table VI, we compare hit ratios of LRU, MQ, and ARC for these workloads. It can be seen that MQ outperforms LRU, while ARC outperforms both MQ and LRU. These results are quite surprising since the algorithm MQ is designed especially for storage servers.

We will show in Figure 7 that ARC can quickly track an evolving workload, namely, P4, that fluctuates from one extreme of recency to the other of frequency. For the trace P4, in Table VII, we compare hit ratios of LRU, MQ, and ARC. It can be clearly seen that ARC outperforms the other two algorithms. LRU is designed for recency while MQ is designed for workloads with stable temporal distance distributions, and, hence, by design, neither can meaningfully track this workload.

Taken together, Tables V, VI and VII imply that ARC is likely to be effective under a wider range of workloads than MQ. Also, while both have constant-time complexity, ARC has a smaller constant, and, hence, less overhead. Finally, adaptation in ARC requires tuning a single scalar, while adaptation in MQ requires maintaining a histogram of observed temporal distances.

V. Experimental Results

A. *Traces*

Table III summarizes various traces that we used in this paper. These traces capture disk accesses by databases, web servers, NT workstations, and a synthetic benchmark for storage controllers. All traces have been filtered by up-stream caches, and, hence, are representative of workloads seen by storage controllers, disks, or RAID controllers.

Trace Name	Number of Requests	Unique Pages
OLTP	914145	186880
P1	32055473	2311485
P2	12729495	913347
P3	3912296	762543
P4	19776090	5146832
P5	22937097	3403835
P6	12672123	773770
P7	14521148	1619941
P8	42243785	977545
P9	10533489	1369543
P10	33400528	5679543
P11	141528425	4579339
P12	13208930	3153310
P13	15629738	2497353
P14	114990968	13814927
ConCat	490139585	47003313
Merge(P)	490139585	47003313
DS1	43704979	10516352
SPC1 like	41351279	6050363
S1	3995316	1309698
S2	17253074	1693344
S3	16407702	1689882
Merge (S)	37656092	4692924

TABLE III. A summary of various traces used in this paper. Number of unique pages in a trace is termed its "footprint".

The trace OLTP has been used in [18], [20], [23]. It contains references to a CODASYL database for a one-hour period. The traces P1–P14 were collected from workstations running Windows NT by using Vtrace which captures disk operations through the use of device filters. The traces were gathered over several months, see [29]. The page size for these traces was 512 bytes. The trace ConCat was obtained by concatenating the traces P1–P14. Similarly, the trace Merge(P) was obtained by merging the traces P1–P14 using time stamps on each of the requests. The idea was to synthesize a trace that may resemble a workload seen by a small storage controller. The trace DS1 was taken off a database server running at a commercial site running an ERP application on top of a commercial database. The trace is seven days long, see [30]. We captured a trace of the SPC1 like synthetic benchmark that contains long sequential scans in addition to random accesses. For precise description of the mathematical algorithms used to generate the benchmark, see [31], [32], [33]. The page size for this trace was 4 KBytes. Finally, we consider three traces S1, S2, and S3 that were disk read accesses initiated by a large commercial search engine in response to various web search requests. The trace S1 was captured over a period of an hour, S2 was captured over roughly four hours, and S3 was captured over roughly six hours. The page size for these traces was 4 KBytes. The trace Merge(S) was obtained by merging the traces S1–S3 using time stamps on each of the requests.

For all traces, we only considered the read requests. All hit ratios reported in this paper are *cold start*. We will report hit ratios in percentages (%).

B. *OLTP*

For this well-studied trace, in Table IV, we compare ARC to a number of algorithms. The tunable parameters for FBR and LIRS were set as in their original papers. The tunable parameters for LRU-2, 2Q, and LRFU were selected in an offline fashion by trying different parameters and selecting the best result for each cache size. The parameter $lifeTime$ of MQ was dynamically estimated, and the history size was set to the cache size resulting in a space overhead comparable to ARC. The algorithm ARC is self-tuning, and requires no user-specified parameters. It can be seen that ARC outperforms all online algorithms, and is comparable to offline algorithms LRU-2, 2Q, and LRFU. The LFU, FBR, LRU-2, 2Q, LRFU, and MIN numbers are exactly the same as those reported in [23].

C. *Two Traces: P8 and P12*

For two traces P8 and P12, in Table V, we display a comparison of hit ratios achieved by ARC with those of LRU, 2Q, LRU-2, LRFU, and LIRS. The tunable parameters for 2Q, LRU-2, LRFU, and LIRS were selected in an offline fashion by trying different parameters and selecting the best result for each trace and each cache size. It can be seen that ARC outperforms LRU and performs close to 2Q, LRU-2, LRFU, and LIRS even when these algorithms use the best offline parameters. While, for brevity, we have exhibited results on only two traces, the same general results continue to hold for all the traces that we examined [3].

D. ARC *and* 2Q

In Table V, we compared ARC with 2Q where the latter used the best fixed, offline choice of its tunable parameter Kin. In Table VI, we compare ARC to 2Q where the latter is also online and is forced to use "reasonable" values of its tunable parameters, specifically, $Kin = 0.3c$ and $Kout = 0.5c$ [20]. It can be seen that ARC outperforms 2Q.

```
ARC($c$)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$.
INITIALIZATION: Set $p = 0$ and set the LRU lists $T_1$, $B_1$, $T_2$, and $B_2$ to empty.

For every $t \geq 1$ and any $x_t$, one and only one of the following four cases must occur.
  Case I: $x_t$ is in $T_1$ or $T_2$. A cache hit has occurred in ARC($c$) and DBL($2c$).
          Move $x_t$ to MRU position in $T_2$.

  Case II: $x_t$ is in $B_1$. A cache miss (resp. hit) has occurred in ARC($c$) (resp. DBL($2c$)).

          $\boxed{\text{ADAPTATION:}}$ Update $p = \min\{p + \delta_1, c\}$ where $\delta_1 = \begin{cases} 1 & \text{if } |B_1| \geq |B_2| \\ |B_2|/|B_1| & \text{otherwise.} \end{cases}$

          REPLACE($x_t, p$). Move $x_t$ from $B_1$ to the MRU position in $T_2$ (also fetch $x_t$ to the cache).

  Case III: $x_t$ is in $B_2$. A cache miss (resp. hit) has occurred in ARC($c$) (resp. DBL($2c$)).

          $\boxed{\text{ADAPTATION:}}$ Update $p = \max\{p - \delta_2, 0\}$ where $\delta_2 = \begin{cases} 1 & \text{if } |B_2| \geq |B_1| \\ |B_1|/|B_2| & \text{otherwise.} \end{cases}$

          REPLACE($x_t, p$). Move $x_t$ from $B_2$ to the MRU position in $T_2$ (also fetch $x_t$ to the cache).

  Case IV: $x_t$ is not in $T_1 \cup B_1 \cup T_2 \cup B_2$. A cache miss has occurred in ARC($c$) and DBL($2c$).

            Case A: $L_1 = T_1 \cup B_1$ has exactly $c$ pages.
                If ($|T_1| < c$)
                    Delete LRU page in $B_1$. REPLACE($x_t, p$).
                else
                    Here $B_1$ is empty. Delete LRU page in $T_1$ (also remove it from the cache).
                endif
            Case B: $L_1 = T_1 \cup B_1$ has less than $c$ pages.
                If ($|T_1| + |T_2| + |B_1| + |B_2| \geq c$)
                    Delete LRU page in $B_2$, if ($|T_1| + |T_2| + |B_1| + |B_2| = 2c$).
                    REPLACE($x_t, p$).
                endif
          Finally, fetch $x_t$ to the cache and move it to MRU position in $T_1$.

Subroutine REPLACE($x_t, p$)
  If ( ($|T_1|$ is not empty) and ( ($|T_1|$ exceeds the target $p$) or ($x_t$ is in $B_2$ and $|T_1| = p$)) )
          Delete the LRU page in $T_1$ (also remove it from the cache), and move it to MRU position in $B_1$.
  else
          Delete the LRU page in $T_2$ (also remove it from the cache), and move it to MRU position in $B_2$.
  endif
```

Fig. 4. Algorithm for Adaptive Replacement Cache. This algorithm is completely self-contained, and can directly be used as a basis for an implementation. No tunable parameters are needed as input to the algorithm. We start from an empty cache and an empty cache directory. ARC corresponds to $T_1 \cup T_2$ and DBL corresponds to $T_1 \cup T_2 \cup B_1 \cup B_2$.

Pages that are discarded from the list L_2 are not put on the Z list. The list Z will have a variable time-dependent size. At any time, Z is the longest list such that (a) $|Z| \leq k$ and (b) the least recent page in Z is more recent than the least recent page in the list L_2. The Z list is related to the LIRS stack in [21].

The Z list can be constructed and used as follows. Whenever the LRU page of L_1 is discarded in Case IV(A) of the Figure 4, make the discarded page the MRU page in the list Z. Discard the LRU page in Z, if $|Z| > k$. Now, whenever the LRU page of L_2 is discarded in Case IV(B) of the Figure 4, ensure that the least recent page in Z is more recent than the *new* least recent page in the list L_2; otherwise, discard pages in the Z list until this condition is satisfied. This latter step may have to discard arbitrarily large number of pages from Z, and, hence the resulting algorithm is constant-time in an expected sense only. Finally, on a hit in the list Z, move the hit page to the top of the list L_2. No adaptation takes place on a hit in Z. We refer to the resulting algorithm as ARC(c, k). In our experiments, we will focus on ARC(c)= ARC($c, 0$).

the list $T_{1,p}$. In light of Remark III.1, the replacement policy is:

B.1 If $|T_{1,p}| > p$, replace the LRU page in $T_{1,p}$.

B.2 If $|T_{1,p}| < p$, replace the LRU page in $T_{2,p}$.

B.3 If $|T_{1,p}| = p$ and the missed page is in $B_{1,p}$ (resp. $B_{2,p}$), replace the LRU page in $T_{2,p}$ (resp. $T_{1,p}$).

The last replacement decision is somewhat arbitrary, and can be made differently if desired. The subroutine REPLACE in Figure 4 is consistent with B.1–B.3.

B. *The Policy*

We now introduce an adaptive replacement policy ARC(c) in the class $\Pi(c)$. At any time, the behavior of the policy ARC is completely described once a certain *adaptation parameter* $p \in [0, c]$ is known. For a given value of p, ARC will behave exactly like FRC$_p$. But, ARC differs from FRC in that it does not use a single fixed value for the parameter p over the entire workload. The policy ARC continuously adapts and tunes p in response to an observed workload.

Let T_1^{ARC}, B_1^{ARC}, T_2^{ARC}, and B_2^{ARC} denote a dynamic partition of L_1 and L_2 corresponding to ARC. For brevity, we will write $T_1 \equiv T_1^{\mathrm{ARC}}$, $T_2 \equiv T_2^{\mathrm{ARC}}$, $B_1 \equiv B_1^{\mathrm{ARC}}$, and $B_2 \equiv B_2^{\mathrm{ARC}}$.

The policy ARC dynamically decides, in response to an observed workload, which item to replace at any given time. Specifically, in light of Remark III.1, on a cache miss, ARC adaptively decides whether to replace the LRU page in T_1 or to replace the LRU page in T_2 depending upon the value of the adaptation parameter p at that time. The intuition behind the parameter p is that it is the *target size* for the list T_1. When p is close to 0 (resp. c), the algorithm can be thought of as favoring L_2 (resp. L_1).

We now exhibit the complete policy ARC in Figure 4. If the two "ADAPTATION" steps are removed, and the parameter p is *a priori* fixed to a given value, then the resulting algorithm is exactly FRC$_p$.

The algorithm in Figure 4 also implicitly simulates the lists $L_1 = T_1 \cup B_1$ and $L_2 = T_2 \cup B_2$. The lists L_1 and L_2 obtained in these fashion are identical to the lists in Figure 1. Specifically, Cases I, II, and III in Figure 4 correspond to Case I in Figure 1. Similarly, Case IV in Figure 4 corresponds to Case II in Figure 1.

C. *Learning*

The policy continually revises the parameter p. The fundamental intuition behind learning is the following: if there is a hit in B_1 then we should increase the size of T_1, and if there is a hit in B_2 then we should increase the size of T_2. Hence, on a hit in B_1, we increase p which is the target size of T_1, and on a hit in B_2, we decrease p. When we increase (resp. decrease) p, we implicitly decrease (resp. increase) $c - p$ which is the target size of T_2. The precise magnitude of the revision in p is also very important. The quantities δ_1 and δ_2 control the magnitude of revision. These quantities are termed as the *learning rates*. The learning rates depend upon the sizes of the lists B_1 and B_2. On a hit in B_1, we increment p by 1, if the size of B_1 is at least the size of B_2; otherwise, we increment p by $|B_2|/|B_1|$. All increments to p are subject to a cap of c. Thus, smaller the size of B_1, the larger the increment. Similarly, On a hit in B_2, we decrement p by 1, if the size of B_2 is at least the size of B_1; otherwise, we decrement p by $|B_1|/|B_2|$. All decrements to p are subject to a minimum of 0. Thus, smaller the size of B_2, the larger the decrement. The idea is to "invest" in the list that is performing the best. The compound effect of a number of such small increments and decrements to p is quite profound as we will demonstrate in the next section. We can roughly think of the learning rule as inducing a "random walk" on the parameter p.

Observe that ARC never becomes complacent and never stops adapting. Thus, if the workload were to suddenly change from a very stable IRM generated to a transient SDD generated one or vice versa, then ARC will track this change and adapt itself accordingly to exploit the new opportunity. The algorithm continuously revises how to invest in L_1 and L_2 according to the recent past.

D. *Scan-Resistant*

Observe that a totally new page, that is, a page that is not in $L_1 \cup L_2$, is always put at the MRU position in L_1. From there it gradually makes it way to the LRU position in L_1. It never affects L_2 unless it is used once again before it is evicted from L_1. Hence, a long sequence of one-time-only reads will pass through L_1 without flushing out possibly important pages in L_2. In this sense, ARC is *scan-resistant*; it will only flush out pages in T_1 and never flush out pages in T_2. Furthermore, when a scan begins, arguably, less hits will be encountered in B_1 compared to B_2, and, hence, by the effect of the learning law, the list T_2 will grow at the expense of the list T_1. This further accentuates the resistance of ARC to scans.

E. *Extra History*

In addition to the c pages in the cache, the algorithm ARC remembers c recently evicted pages. An interesting question is whether we can incorporate more history information in ARC to further improve it. Let us suppose that we are allowed to remember k extra pages. We now demonstrate how to carry out this step. Define a LRU list Z that contains pages that are discarded from the list L_1 in Case IV(A) of the algorithm in Figure 4.

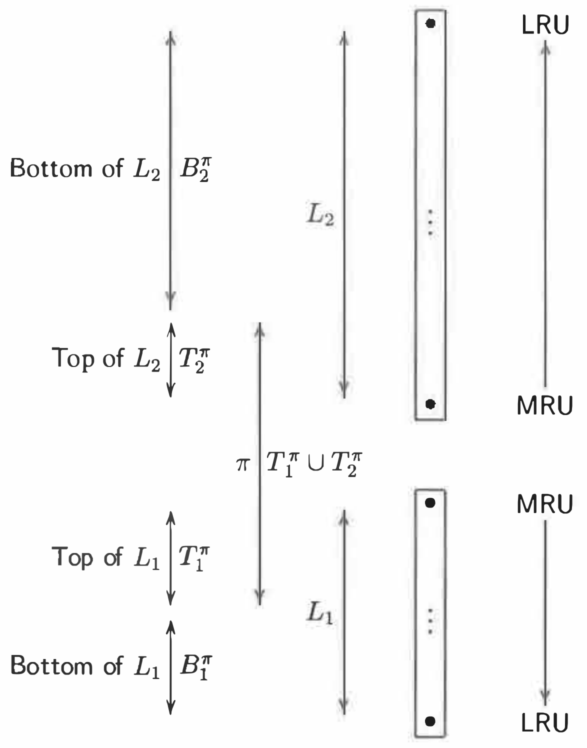

LRU
Bottom of L_2 B_2^π
L_2
Top of L_2 T_2^π
MRU
π $T_1^\pi \cup T_2^\pi$
MRU
Top of L_1 T_1^π
L_1
Bottom of L_1 B_1^π
LRU

Fig. 3. General structure of a generic cache replacement policy $\pi(c) \in \Pi(c)$. The lists L_1 and L_2 are exactly as in Figure 2. The list L_1 is partitioned into a top portion T_1^π and a bottom portion B_1^π. Similarly, the list L_2 is partitioned into a top portion T_2^π and a bottom portion B_2^π. The policy $\pi(c)$ maintains pages in $T_1^\pi \cup T_2^\pi$ in cache.

A.1 The lists T_1^π and B_1^π are disjoint and, also, the lists T_2^π and B_2^π are disjoint, and

$$L_1 = T_1^\pi \cup B_1^\pi \text{ and } L_2 = T_2^\pi \cup B_2^\pi.$$

A.2 If $|L_1 \cup L_2| < c$, then both B_1^π and B_2^π are empty.

A.3 If $|L_1 \cup L_2| \geq c$, then, together, T_1^π and T_2^π contain exactly c pages.

A.4 Either T_1^π (resp. T_2^π) or B_1^π (resp. B_2^π) is empty, or the LRU page in T_1^π (resp. T_2^π) is more recent than the MRU page in B_1^π (resp. B_2^π).

A.5 For all traces and at each time, $T_1^\pi \cup T_2^\pi$ will contain exactly those pages that would be maintained in cache by the policy $\pi(c)$.

The generic structure of the policy π is depicted in Figure 3. It follows from Conditions A.1 and A.5 that the pages contained in a cache of size c managed by $\pi(c)$ will always be a subset of the pages managed by DBL$(2c)$. Since any policy in $\Pi(c)$ must track all pages that would have been in DBL$(2c)$, it would require essentially double the space overhead of LRU.

Remark III.1 Condition A.4 implies that if a page in L_1 (resp. L_2) is kept, then all pages in L_1 (resp. L_2) that are more recent than it must all be kept in the cache. Hence, the policy $\pi(c)$ can be thought of as "skimming the top (or most recent) few pages" in L_1 and L_2. Suppose that we are managing a cache with $\pi(c) \in \Pi(c)$, and also let us suppose that the cache is full, that is, $|T_1^\pi \cup T_2^\pi| = c$, then, it follows from Condition A.4 that, from now on, for any trace, on a cache miss, only two *actions* are available to the policy $\pi(c)$: (i) replace the LRU page in T_1^π or (ii) replace the LRU page in T_2^π.

C. LRU

We now show that the policy LRU(c) is contained in the class $\Pi(c)$. In particular, we show that the most recent c pages will always be contained in DBL$(2c)$. To see this fact observe that DBL$(2c)$ deletes either the LRU item in L_1 or the LRU item in L_2. In the first case, L_1 must contain exactly c items (see case II(A) in Figure 1). In the second case, L_2 must contain at least c items (see case II(B)(1) in Figure 1). Hence, DBL never deletes any of the most recently seen c pages and always contains all pages contained in a LRU cache with c items. It now follows that there must exist a dynamic partition of lists L_1 and L_2 into lists T_1^{LRU}, B_1^{LRU}, T_2^{LRU}, and B_2^{LRU}, such that the conditions A.1–A.5 hold. Hence, the policy LRU(c) is contained in the class $\Pi(c)$ as claimed.

Conversely, if we consider DBL$(2c')$ for some positive integer $c' < c$, then the most recent c pages need not always be in DBL$(2c')$. For example, consider the trace $1, 2, \ldots, c, 1, 2, \ldots, c, \ldots, 1, 2, \ldots, c, \ldots$. For this trace, hit ratio of LRU(c) approaches 1 as size of the trace increases, but hit ratio of DBL$(2c')$, for any $c' < c$, is zero. The above remarks shed light on choice of $2c$ as the size of the cache directory DBL.

IV. Adaptive Replacement Cache

A. Fixed Replacement Cache

We now describe a *fixed replacement cache*, FRC$_p(c)$, in the class $\Pi(c)$, where p is a tunable parameter p, $0 \leq p \leq c$. The policy FRC$_p(c)$ will satisfy conditions A.1–A.5. For brevity, let us write $T_{1,p} \equiv T_1^{\mathrm{FRC}_p(c)}$, $T_{2,p} \equiv T_2^{\mathrm{FRC}_p(c)}$, $B_{1,p} \equiv B_1^{\mathrm{FRC}_p(c)}$, and $B_{2,p} \equiv B_2^{\mathrm{FRC}_p(c)}$.

The crucial additional condition that FRC$_p(c)$ satisfies is that it will attempt to keep exactly p pages in the list $T_{1,p}$ and exactly $c - p$ pages in the list $T_{2,p}$. In other words, the policy FRC$_p(c)$ attempts to keep exactly p top (most recent) pages from the list L_1 and $c - p$ top (most recent) pages from the list L_2 in the cache. Intuitively, the parameter p is the *target size* for

III. A Class of Replacement Policies

Let c denote the cache size in number of pages. For a cache replacement policy π, we will write $\pi(c)$ when we want to emphasize the number of pages being managed by the policy.

We will first introduce a policy DBL$(2c)$ that will manage and remember twice the number of pages present in the cache. With respect to the policy DBL, we introduce a new class of cache replacement policies $\Pi(c)$.

A. Double Cache and a Replacement Policy

Suppose we have a cache that can hold $2c$ pages. We now describe a cache replacement policy DBL$(2c)$ to manage such a cache. We will use this construct to motivate the development of an adaptive cache replacement policy for a cache of size c.

The cache replacement policy DBL$(2c)$ maintains two variable-sized lists L_1 and L_2, the first containing pages that have been seen *only once recently* and the second containing pages that have been seen *at least twice recently*. Precisely, a page is in L_1 if has been requested exactly once since the last time it was removed from $L_1 \cup L_2$, or if it was requested only once and was never removed from $L_1 \cup L_2$. Similarly, a page is in L_2 if it has been requested more than once since the last time it was removed from $L_1 \cup L_2$, or was requested more than once and was never removed from $L_1 \cup L_2$. The replacement policy is:

> Replace the LRU page in L_1, if L_1 contains exactly c pages; otherwise, replace the LRU page in L_2.

The replacement policy attempts to equalize the sizes of two lists. We exhibit a complete algorithm that captures DBL in Figure 1 and pictorially illustrate its structure in Figure 2. The sizes of the two lists can fluctuate, but the algorithm ensures that following invariants will always hold:

$$0 \le |L_1| + |L_2| \le 2c,\ 0 \le |L_1| \le c,\ 0 \le |L_2| \le 2c.$$

B. A New Class of Policies

We now propose a new class of policies $\Pi(c)$. Intuitively, the proposed class will contain demand paging policies that track all $2c$ items that would have been in a cache of size $2c$ managed by DBL, but physically keep only (at most) c of those in the cache at any given time.

Let L_1 and L_2 be the lists associated with DBL$(2c)$. Let $\Pi(c)$ denote a class of demand paging cache replacement policies such that for every policy $\pi(c) \in \Pi(c)$ there exists a dynamic partition of list L_1 into a top portion T_1^π and a bottom portion B_1^π and a dynamic partition of list L_2 into a top portion T_2^π and a bottom portion B_2^π such that

```
DBL(2c)

INPUT: The request stream $x_1, x_2, \ldots, x_t, \ldots$
INITIALIZATION: Set $\ell_1 = 0$, $\ell_2 = 0$, $L_1 = \emptyset$ and $L_2 = \emptyset$.

For every $t \ge 1$ and any $x_t$, one and only one of the following two cases must occur.
  Case I: $x_t$ is in $L_1$ or $L_2$.
      A cache hit has occurred. Make $x_t$ the MRU page in $L_2$.
  Case II: $x_t$ is neither in $L_1$ nor in $L_2$.
      A cache miss has occurred. Now, one and only one of the two cases must occur.
        Case A: $L_1$ has exactly $c$ pages.
            Delete the LRU page in $L_1$ to make room for the new page, and make $x_t$ the MRU page in $L_1$.
        Case B: $L_1$ has less than $c$ pages.
            1) If the cache is full, that is, $(|L_1| + |L_2|) = 2c$, then delete the LRU page in $L_2$ to make room for the new page.
            2) Insert $x_t$ as the MRU page in $L_1$.
```

Fig. 1. Algorithm for the cache replacement policy DBL that manages $2c$ pages in cache.

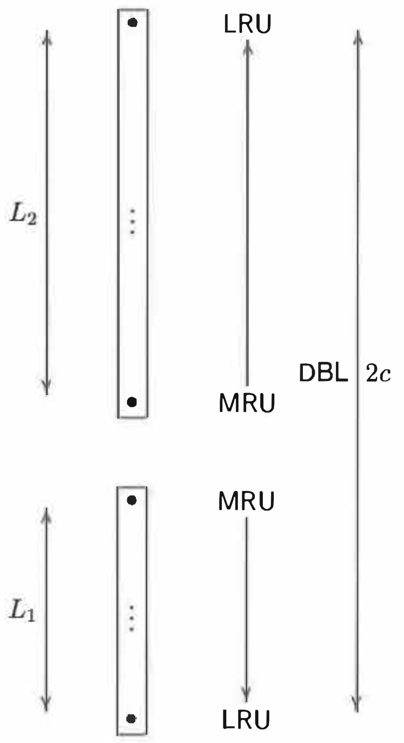


Fig. 2. General structure of the cache replacement policy DBL. The cache is partitioned into two LRU lists: L_1 and L_2. The former contains pages that have been seen only once "recently" while the latter contains pages that have been seen at least twice "recently". Visually, the list L_2 is inverted when compared to L_1. This allows us to think of the more recent items in both the lists as closer in the above diagram. The replacement policy deletes the LRU page in L_1, if L_1 contains exactly c pages; otherwise, it replaces the LRU page in L_2.

smoothing that is widely used in statistics. The LRFU policy is to replace the page with the smallest $C(x)$ value. Intuitively, as λ approaches 0, the C value is simply the number of occurrences of page x and LRFU collapses to LFU. As λ approaches 1, due to the exponential decay, the C value emphasizes recency and LRFU collapses to LRU. The performance of the algorithm depends crucially on the choice of λ, see [23, Figure 7]. A later adaptive version, namely, Adaptive LRFU (ALRFU) dynamically adjusts the parameter λ [24]. Still, LRFU has two fundamental limitations that hinder its use in practice: (i) LRFU and ALRFU both require an additional tunable parameter for controlling correlated references. The choice of this parameter matters, see [23, Figure 8]. (ii) The implementation complexity of LRFU fluctuates between constant per request to logarithmic in cache size per request. However, due to multiplications and exponentiations, its practical complexity is significantly higher than that of even LRU-2, see, Table I. It can be seen that, for small values of λ, LRFU can be as much as 50 times slower than LRU and ARC. Such overhead can potentially wipe out the entire benefit of a higher hit ratio.

E. Temporal Distance Distribution

Recently, [25] studied a multi-queue replacement policy MQ. The idea is to use m (typically, $m = 8$) LRU queues: $Q_0, \ldots, Q_{m-1}$, where Q_i contains pages that have been seen at least 2^i times but no more than $2^{i+1} - 1$ times recently. The algorithm also maintains a history buffer Q_{out}. On a page hit, the page frequency is incremented, the page is placed at the MRU position of the appropriate queue, and its $expireTime$ is set to $currentTime + lifeTime$, where $lifeTime$ is a tunable parameter. On each access, $expireTime$ for the LRU page in each queue is checked, and if it is less than $currentTime$, then the page is moved to the MRU position of the next lower queue. The optimal values of $lifeTime$ and the length of Q_{out} depend upon the workload and the cache size.

The algorithm MQ is designed for storage controllers. Due to up-stream caches, two consecutive accesses to a single page have a relatively long temporal distance. The algorithm assumes that the temporal distance possesses a "hill" shape. In this setting, the recommended value of the $lifeTime$ parameter is the peak temporal distance, and can be dynamically estimated for each workload.

The algorithm ARC makes no such stringent assumption about the shape of the temporal distance distribution, and, hence, is likely to be robust under a wider range of workloads. Also, MQ will adjust to workload evolution when a measurable change in peak temporal distance can be detected, whereas ARC will track an evolving workload nimbly since it adapts continually. While MQ has constant-time overhead, it still needs to check time stamps of LRU pages for m queues on every request, and, hence, has a higher overhead than LRU, ARC, and 2Q. For example, in the context of Table I, with $m = 8$, the overhead ranged from 36 to 54 seconds for various cache sizes.

F. Caching using Multiple Experts

Recently, [26] proposed a master-policy that simulates a number of caching policies (in particular, 12 policies) and, at any given time, adaptively and dynamically chooses one of the competing policies as the "winner" and switches to the winner. As they have noted, "rather than develop a new caching policy", they select the best policy amongst various competing policies. Since, ARC (or any of the other policies discussed above) can be used as one of the competing policies, the two approaches are entirely complementary. From a practical standpoint, a limitation of the master-policy is that it must simulate all competing policies, and, hence, requires high space and time overhead. Recently, [27] applied above ideas to distributed caching.

G. Ghost Caches

In this paper, we will maintain a larger cache directory than that is needed to support the underlying cache. Such directories are known as a *shadow cache* or as a *ghost cache*. Previously, ghost caches have been employed in a number of cache replacement algorithms such as 2Q, MQ, LRU-2, ALRFU, and LIRS to remember recently evicted cache pages. Most recently, while studying how to make a storage array's cache more exclusive of the client caches, [28] used ghost caches to simulate two LRU lists: one for disk-read blocks and the other for client-demoted blocks. The hits rates on the ghost LRU lists were then used to adaptively determine the suitable insertion points for each type of data in a LRU list.

H. Summary

In contrast to LRU-2, 2Q, LIRS, FBR, and LRFU which require offline selection of tunable parameters, our replacement policy ARC is online and is completely self-tuning. Most importantly, ARC is empirically universal. The policy ARC maintains no frequency counts, and, hence, unlike LFU and FBR, it does not suffer from periodic rescaling requirements. Also, unlike LIRS, the policy ARC does not require potentially unbounded space overhead. In contrast to MQ, the policy ARC may be useful for a wider range of workloads, adapts quickly to evolving workloads, and has less computational overhead. Finally, ARC, 2Q, LIRS, MQ, and FBR have constant-time implementation complexity while LFU, LRU-2, and LRFU have logarithmic implementation complexity.

	CIP/c								
c	0.01	0.05	0.1	0.25	0.5	0.75	0.9	0.95	0.99
1024	0.87	1.01	1.41	3.03	3.77	4.00	$\boxed{4.07}$	$\boxed{4.07}$	$\boxed{4.07}$
2048	1.56	2.08	3.33	4.32	4.62	4.77	4.80	$\boxed{4.83}$	$\boxed{4.83}$
4096	2.94	4.45	5.16	5.64	$\boxed{5.81}$	5.79	5.70	5.65	5.61
8192	5.36	7.02	7.39	$\boxed{7.54}$	7.36	6.88	6.47	6.36	6.23
16384	9.53	10.55	$\boxed{10.67}$	10.47	9.47	8.38	7.60	7.28	7.15
32768	15.95	$\boxed{16.36}$	16.23	15.32	13.46	11.45	9.92	9.50	9.03
65536	$\boxed{25.79}$	25.66	25.12	23.64	21.33	18.26	15.82	14.99	14.58
131072	$\boxed{39.58}$	38.88	38.31	37.46	35.15	32.21	30.39	29.79	29.36
262144	$\boxed{53.43}$	53.04	52.99	52.09	51.73	49.42	48.73	49.20	49.11
524288	$\boxed{63.15}$	63.14	62.94	62.98	62.00	60.75	60.40	60.57	60.82

TABLE II. Hit ratios (in percentages) achieved by the algorithm LRU-2 on a trace P12 for various values of the tunable parameter CIP and various cache sizes. The trace P12 was collected from a workstation running Windows NT by using Vtrace which captures disk requests, for details of the trace, see Section V-A. The cache size c represents number of 512 byte pages.

and the cache size. After theoretically analyzing how to set parameter $Kout$, [20] concluded that "Since this formula requires an a priori estimate of the miss rate, it is of little use in practical tuning." They suggested $Kout = 0.5c$ "is almost always a good choice".

Another recent algorithm is Low Inter-reference Recency Set (LIRS) [21]. The algorithm maintains a variable size LRU stack whose LRU page is the L_{lirs}-th page that has been seen at least twice recently, where L_{lirs} is a parameter. From all the pages in the stack, the algorithm keeps all the L_{lirs} pages that have been seen at least twice recently in the cache as well as L_{hirs} pages that have been seen only once recently. The parameter L_{hirs} is similar to CIP of LRU-2 or Kin of 2Q. The authors suggest setting L_{hirs} to 1% of the cache size. This choice will work well for stable workloads drawn according to the IRM, but not for those LRU-friendly workloads drawn according to the SDD. Just as CIP affects LRU-2 and Kin affects 2Q, we have found that the parameter L_{hirs} crucially affects LIRS. A further limitation of LIRS is that it requires a certain "stack pruning" operation that in the worst case may have to touch a very large number of pages in the cache. This implies that overhead of LIRS is constant-time in the expected sense, and not in the worst case as for LRU. Finally, the LIRS stack may grow arbitrarily large, and, hence, it needs to be *a priori* limited.

Our idea of separating items that have been only seen once recently from those that have been seen at least twice recently is related to similar ideas in LRU-2, 2Q, and LIRS. However, the precise structure of our lists L_1 and L_2 and the self-tuning, adaptive nature of our algorithm have no analogue in these papers.

D. Recency and Frequency

Over last few years, interest has focussed on combining recency and frequency. Several papers have attempted to bridge the gap between LRU and LFU by combining recency and frequency in various ways. We shall mention two heuristic algorithms in this direction.

Frequency-based replacement (FBR) policy [22] maintains a LRU list, but divides it into three sections: new, middle, and old. For every page in cache, a counter is also maintained. On a cache hit, the hit page is moved to the MRU position in the new section; moreover, if the hit page was in the middle or the old section, then its reference count is incremented. The key idea known as *factoring out locality* was that if the hit page was in the new section then the reference count is not incremented. On a cache miss, the page in the old section with the smallest reference count is replaced. A limitation of the algorithm is that to prevent cache pollution due to stale pages with high reference count but no recent usage the algorithm must periodically resize (rescale) all the reference counts. The algorithm also has several tunable parameters, namely, the sizes of all three sections, and some other parameters C_{max} and A_{max} that control periodic resizing. Once again, much like in LRU-2 and 2Q, different values of the tunable parameters may be suitable for different workloads or for different cache sizes. The historical importance of FBR stems from the fact it was one of the earliest papers to combine recency and frequency. It has been shown that performance of FBR is similar to that of LRU-2, 2Q, and LRFU [23].

Recently, a class of policies, namely, Least Recently/Frequently Used (LRFU), that subsume LRU and LFU was studied [23]. Initially, assign a value $C(x) = 0$ to every page x, and, at every time t, update as:

$$C(x) = \begin{cases} 1 + 2^{-\lambda}C(x) & \text{if } x \text{ is referenced at time } t; \\ 2^{-\lambda}C(x) & \text{otherwise,} \end{cases}$$

where λ is a tunable parameter. With hindsight, we can easily recognize the update rule as a form of *exponential*

c	LRU	ARC	2Q	LRU-2	LRFU		
					λ		
					10^{-7}	10^{-3}	.99
1024	17	14	17	33	554	408	28
2048	12	14	17	27	599	451	28
4096	12	15	17	27	649	494	29
8192	12	16	18	28	694	537	29
16384	13	16	19	30	734	418	30
32768	14	17	18	31	716	420	31
65536	14	16	18	32	648	424	34
131072	14	15	16	32	533	432	39
262144	13	13	14	30	427	435	42
524288	12	13	13	27	263	443	45

TABLE I. A comparison of computational overhead of various cache algorithms on a trace P9 that was collected from a workstation running Windows NT by using Vtrace which captures disk requests. For more details of the trace, see Section V-A. The cache size c represents number of 512 byte pages. To obtain the numbers reported above, we assumed that a miss costs nothing more than a hit. This focuses the attention entirely on the "book-keeping" overhead of the cache algorithms. All timing numbers are in seconds, and were obtained by using the "clock()" subroutine in "time.h" of the GNU C compiler. It can be seen that the computational overhead of ARC and 2Q is essentially the same as that of LRU. It can also be seen that LRU-2 has roughly double the overhead of LRU, and that LRFU can have very large overhead when compared to LRU. The same general results hold for all the traces that we examined.

the workload or the request stream is drawn from a LRU Stack Depth Distribution (SDD), then LRU is the optimal policy [16]. LRU has several advantages, for example, it is simple to implement and responds well to changes in the underlying SDD model. However, while the SDD model captures "recency", it does not capture "frequency". To quote from [16, p. 282]: "The significance of this is, in the long run, that each page is equally likely to be referenced and that therefore the model is useful for treating the clustering effect of locality but not the nonuniform page referencing."

C. *Frequency*

The Independent Reference Model (IRM) provides a workload characterization that captures the notion of frequency. Specifically, IRM assumes that each page reference is drawn in an independent fashion from a fixed distribution over the set of all pages in the auxiliary memory. Under the IRM model, policy LFU that replaces the least frequently used page is known to be optimal [16], [17]. The LFU policy has several drawbacks: it requires logarithmic implementation complexity in cache size, pays almost no attention to recent history, and does not adapt well to changing access patterns since it accumulates stale pages with high frequency counts that may no longer be useful.

A relatively recent algorithm LRU-2 [18], [19] approximates LFU while eliminating its lack of adaptivity to the evolving distribution of page reference frequencies. This was a significant practical step forward. The basic idea is to remember, for each page, the last two times when it was requested, and to replace the page with the least recent penultimate reference. Under the IRM assumption, it is known that LRU-2 has the largest expected hit ratio of any on-line algorithm that knows at most two most recent references to each page [19]. The algorithm has been shown to work well on several traces [18], [20]. Nonetheless, LRU-2 still has two practical limitations [20]: (i) it needs to maintain a priority queue, and, hence, it requires logarithmic implementation complexity and (ii) it contains at one crucial tunable parameter, namely, *Correlated Information Period* (CIP), that roughly captures the amount of time a page that has only been seen once recently should be kept in the cache.

In practice, logarithmic implementation complexity is a severe overhead, see, Table I. This limitation was mitigated in 2Q [20] which reduces the implementation complexity to constant per request. The algorithm 2Q uses a simple LRU list instead of the priority queue used in LRU-2; otherwise, it is similar to LRU-2. Policy ARC has a computational overhead similar to 2Q and both are better than LRU-2, see, Table I.

Table II shows that the choice of the parameter CIP crucially affects performance of LRU-2. It can be seen that no single fixed *a priori* choice works uniformly well across across various cache sizes, and, hence, judicious selection of this parameter is crucial to achieving good performance. Furthermore, we have found that no single *a priori* choice works uniformly well across across various workloads and cache sizes that we examined. For example, a very small value for the CIP parameters work well for stable workloads drawn according to the IRM, while a larger value works well for workloads drawn according to the SDD. Indeed, it has been previously noted [20] that "it was difficult to model the tunables of the algorithm exactly." This underscores the need for on-line, on-the-fly adaptation.

Unfortunately, the second limitation of LRU-2 persists even in 2Q. The authors introduce two parameters (Kin and $Kout$) and note that "Fixing these parameters is potentially a tuning question ..." [20]. The parameter Kin is essentially the same as the parameter CIP in LRU-2. Once again, it has been noted [21] that "Kin and $Kout$ are predetermined parameters in 2Q, which need to be carefully tuned, and are sensitive to types of workloads." Due to space limitation, we have shown Table II only for LRU-2, however, we have observed similar dependence of 2Q on the workload over such access patterns. We seek a cache replacement policy that will adapt in an on-line, on-the-fly fashion to such dynamically evolving workloads.

We propose a new cache replacement policy, namely, Adaptive Replacement Cache (ARC). The basic idea behind ARC is to maintain two LRU lists of pages. One list, say L_1, contains pages that have been seen only once "recently", while the other list, say L_2, contains pages that have been seen at least twice "recently". The items that have been seen twice within a short time have a low inter-arrival rate, and, hence, are thought of as "high-frequency". Hence, we think of L_1 as capturing "recency" while L_2 as capturing "frequency". We endeavor to keep these two lists to roughly the same size, namely, the cache size c. Together the two lists remember exactly twice the number of pages that would fit in the cache. In other words, ARC maintains a *cache directory* that remembers twice as many pages as in the cache memory. At any time, ARC chooses a variable number of most recent pages to keep from L_1 and from L_2. The precise number of pages drawn from the list L_1 is a tunable parameter that is adaptively and continually tuned. Let FRC_p denote a *fixed replacement policy* that attempts to keep the p most recent pages in L_1 and $c-p$ most recent pages in L_2 in cache at all times. At any given time, the policy ARC behaves like FRC_p for some fixed p. However, ARC may behave like FRC_p at one time and like FRC_q, $p \neq q$, at some other time. The key new idea is to adaptively–in response to an evolving workload–decide how many top pages from each list to maintain in the cache at any given time. We achieve such on-line, on-the-fly adaptation by using a *learning rule* that allows ARC to track a workload quickly and effectively. The effect of the learning rule is to induce a "random walk" on the parameter p. Intuitively, by learning from the recent past, ARC attempts to keep those pages in the cache that have the greatest likelihood of being used in the near future. It acts as a filter to detect and track temporal locality. If during some part of the workload, recency (resp. frequency) becomes important, then ARC will detect the change, and configure itself to exploit the opportunity. We think of ARC as *dynamically, adaptively, and continually* balancing between recency and frequency–in an *online* and *self-tuning* fashion–in response to evolving and possibly changing access patterns.

We empirically demonstrate that ARC works as well as the policy FRC_p that assigns a fixed portion of the cache to recent pages and the remaining fixed portion to frequent pages–even when the latter uses the best fixed, offline workload and cache size dependent choice for the parameter p. In this sense, ARC is *empirically universal* [1]. Surprisingly, ARC–which is completely online–delivers performance comparable to LRU-2, 2Q, LRFU, and LIRS–even when these policies use the best tuning parameters that were selected in an offline fashion. The policy ARC also compares favorably to an online adaptive policy MQ.

To implement ARC, we need two LRU lists. Hence, ARC is no more difficult to implement than LRU, has constant-time complexity per request, and requires only marginally higher space overhead over LRU. In a real-life implementation, we found that the space overhead of ARC was 0.75% of the cache size. We say that ARC is *low overhead*. In contrast, LRU-2 and LRFU require logarithmic time complexity per request. As a result, in all our simulations, LRU-2 was a factor of 2 slower than ARC and LRU, while LRFU can be as much as a factor of 50 slower than ARC and LRU.

The policy ARC is *scan-resistant*, that is, it allows one-time-only sequential read requests to pass through the cache without flushing pages that have temporal locality. By the same argument, it effectively handles long periods of low temporal locality.

Finally, on a large number of real life workloads drawn from CODASYL, 14 workstation disk drives, a commercial ERP system, a SPC1 like [2] synthetic benchmark, as well as web search requests, we demonstrate that ARC substantially outperforms LRU. As anecdotal evidence, for a workstation disk drive workload, at 16MB cache, LRU delivers a hit ratio of 4.24% while ARC achieves a hit ratio of 23.82%, and, for a SPC1 like benchmark, at 4GB cache, LRU delivers a hit ratio of 9.19% while ARC achieves a hit ratio of 20%.

C. A Brief Outline of the Paper

In Section II, we briefly review relevant work and provide a context for ARC. In Section III, we introduce a class of replacement policies and show that this class contains LRU as a special case. In Section IV, we introduce the policy ARC. In Section V, we present experimental results on several workloads. Finally, in Section VI, we present conclusions.

II. Prior Work: A Brief Review

A. Offline Optimal

For an *a priori* known page reference stream, Belady's MIN that replaces the page that has the greatest forward distance is known to be optimal in terms of the hit ratio [12], [13]. The policy MIN provides an upper bound on the achievable hit ratio by any on-line policy.

B. Recency

The policy LRU always replaces the least recently used page [13]. It dates back at least to 1965 [14], and may in fact be older. Various approximations and improvements to LRU abound, see, for example, enhanced clock algorithm [15]. It is known that if

ARC: A SELF-TUNING, LOW OVERHEAD REPLACEMENT CACHE

Nimrod Megiddo and Dharmendra S. Modha
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
Email: {megiddo,dmodha}@almaden.ibm.com

Abstract—We consider the problem of cache management in a demand paging scenario with uniform page sizes. We propose a new cache management policy, namely, Adaptive Replacement Cache (ARC), that has several advantages.

In response to evolving and changing access patterns, ARC *dynamically, adaptively, and continually* balances between the recency and frequency components in an *online* and *self-tuning* fashion. The policy ARC uses a learning rule to adaptively and continually revise its assumptions about the workload.

The policy ARC is *empirically universal*, that is, it empirically performs as well as a certain *fixed replacement policy*– even when the latter uses the best workload-specific tuning parameter that was selected in an offline fashion. Consequently, ARC works uniformly well across varied workloads and cache sizes without any need for workload specific *a priori* knowledge or tuning. Various policies such as LRU-2, 2Q, LRFU, and LIRS require user-defined parameters, and, unfortunately, no single choice works uniformly well across different workloads and cache sizes.

The policy ARC is simple-to-implement and, like LRU, has constant complexity per request. In comparison, policies LRU-2 and LRFU both require logarithmic time complexity in the cache size.

The policy ARC is *scan-resistant*: it allows one-time sequential requests to pass through without polluting the cache.

On 23 real-life traces drawn from numerous domains, ARC leads to substantial performance gains over LRU for a wide range of cache sizes. For example, for a SPC1 like synthetic benchmark, at 4GB cache, LRU delivers a hit ratio of 9.19% while ARC achieves a hit ratio of 20%.

I. INTRODUCTION

"We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, J. von Neumann, in *Preliminary Discussion of the Logical Design of Electronic Computing Instrument*, Part I, Vol. I, Report prepared for U.S. Army Ord. Dept., 28 June 1946.

A. The Problem

Caching is one of the oldest and most fundamental metaphor in modern computing. It is widely used in storage systems (for example, IBM ESS or EMC Symmetrix), databases [1], web servers [2], middleware [3], processors [4], file systems [5], disk drives [6], RAID controllers [7], operating systems [8], and in varied and numerous other applications such as data compression [9] and list updating [10]. Any substantial progress in caching algorithms will affect the entire modern computational stack.

Consider a system consisting of two memory levels: *main* (or *cache*) and *auxiliary*. The cache is assumed to be significantly faster than the auxiliary memory, but is also significantly more expensive. Hence, the size of the cache memory is usually only a fraction of the size of the auxiliary memory. Both memories are managed in units of uniformly sized items known as *pages*. We assume that the cache receives a continuous *stream of requests* for pages. We assume a *demand paging* scenario where a page of memory is *paged in* the cache from the auxiliary memory only when a request is made for the page and the page is not already present in the cache. In particular, demand paging rules out pre-fetching. For a full cache, before a new page can be brought in one of the existing pages must be *paged out*. The victim page is selected using a cache *replacement policy*. Under demand paging model, the replacement policy is the only algorithm of interest. The most important metric for a cache replacement policy is its *hit rate*–the fraction of pages that can be served from the main memory. The *miss rate* is the fraction of pages that must be paged into the cache from the auxiliary memory. Another important metric for a cache replacement policy is its overhead which should be low.

The problem of cache management is to design a replacement policy that maximizes the hit rate measured over a very long trace subject to the important practical constraints of minimizing the computational and space overhead involved in implementing the policy.

B. Our Contributions

One of the main themes of this paper is to design a replacement policy with a high hit ratio while paying conscientious attention to its implementation complexity. Another equally important theme of this paper is that real-life workloads possess a great deal of richness and variation, and do not admit a one-size-fits-all characterization. They may contain long sequential I/Os or moving hot spots. The frequency and scale of temporal locality may also change with time. They may fluctuate between stable, repeating access patterns and access patterns with transient clustered references. No static, *a priori* fixed replacement policy will work well

References

[1] M. C. Abraham, H. Schmidt, R. J. Ram, T. A. Savas, H. I. Smith, M. Hwang, and C. Ross. MFM studies of nanomagnetic arrays. http://rleweb.mit.edu/sclaser/mmm99/sld001.htm, Massachusetts Institute of Technology, Seminconductor Laser Group, Feb 2000. Accessed Dec. 2002.

[2] L.R. Carley, J.A. Bain, and G.K. Fedder. Single chip computers with MEMS-based magnetic memory. In *44th Annual Conference on Magnetism and Magnetic Materials*, November 1999.

[3] Center for highly integrated information processing and storage systems (CHIPS) home page. http://www.chips.ece.cmu.edu, Carnegie Institute of Technology, Carnegie Mellon University. Accessed Dec. 2002.

[4] S. H. Charap, Pu Ling Lu, and Yanjun He. Thermal stability of recorded information at high densities. In *IEEE Transactions on Magnetics*, volume 33, pages 978–983, January 1997.

[5] M. Despont, J. Brugger, U. Drechsler, U. Dürig, W. Häberle, M. Lutwyche, H. Rothuizen, R. Stutz, R. Widmer, H. Rohrer, G. Binnig, and P. Vettiger. VLSI-NEMS chip for AFM data storage. In *Technical Digest. Twelfth IEEE International Conference on Micro Electro Mechanical Systems*, pages 564–69, Orlando, FL, January 1999.

[6] J. L. Griffin, S. W.Schlosser, G. R Ganger, and D. F. Nagle. Modeling and performance of MEMS-based storage devices. In *Proceedings of ACM SIGMETRICS 2000*, pages 56–65, (Santa Clara, CA), June 2000.

[7] J. L. Griffin, S. W.Schlosser, G. R Ganger, and D. F. Nagle. Operating system management of MEMS-based storage devices. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 227–242, (San Diego, California, USA), October 2000.

[8] J. L. Hennessy and D. A.Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, 1996.

[9] Ying Lin, Scott A. Brandt, Darrell D. E. Long, and Ethan L. Miller. Power conservation strategies for MEMS-based storage devices. In *Proceedings of the 10th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '02)*, Fort Worth, TX, October 2002.

[10] Tara M. Madhyastha and Katherine Pu Yang. Physical modeling of probe-based storage. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems*, pages 207–224, April 2001.

[11] H. J. Mamin, B. D. Terris, L. S. Fan, S. Hoen, R. C. Barrett, and D. Rugar. High-density data storage using proximal probe techniques. *IBM Journal of Research and Development*, 39(6):681–99, November 1995.

[12] G. Moore. Progress in digital integrated electronics. In *Proceedings of the IEEE Digital Integrated Electronic Device Meeting*, pages 11–13, 1975.

[13] Personal communication. Dr. Charles C. Morehouse, HP laboratories, May 2001.

[14] Demetri Psaltis and Geoffrey W. Burr. Holographic data storage. *IEEE Computer*, 31(2):52–60, February 1998.

[15] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *Proceedings of Winter'93 USENIX Conference*, pages 405–420, January 1993.

[16] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[17] S. W. Schlosser, J. L. Griffin, D. F. Nagle, and G. R. Ganger. Filling the memory access gap: A case for on-chip magnetic storage. Technical Report CMU–CS–99–174, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, November 1999.

[18] Steven W. Schlosser, John Linwood Griffin, David Nagle, and Gregory R. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, Boston, Massachusetts, November 2000.

[19] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead cach es and request reordering. In *Proceedings of ACM SIGMETRICS 1998*, pages 183–191, Madison, WI, June 1998.

[20] Glenn T. Sincerbox, editor. *Selected Papers on Holographic Storage*, volume MS 95 of *SPIE Milestone series.* International Society for Optical Engineering, 1994.

[21] Miriam Sivan-Zimet. Workload based optimization of probe-based storage. Technical Report UCSC–CRL–01–06, University of California, Santa Cruz, July 2001. http://www.cse.ucsc.edu/t̃ara/papers/miri_thesis.pdf.

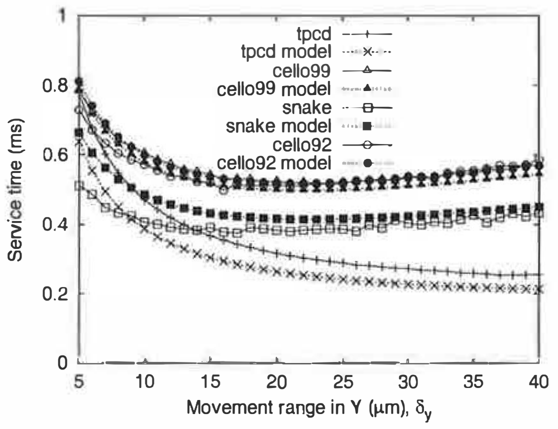
[22] Miriam Sivan-Zimet and Tara M. Madhyastha. Workload based optimization of probe-based storage. In *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 256–257, Marina Del Ray, CA, June 2002.

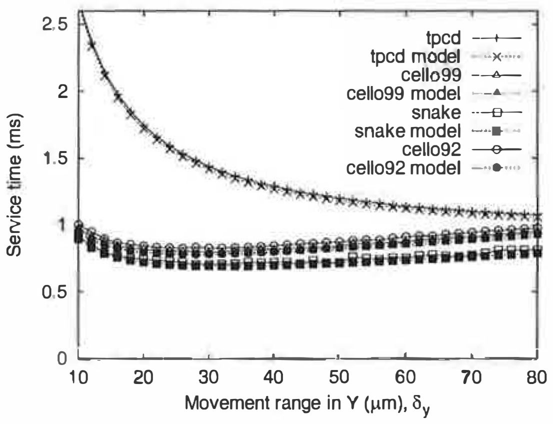
[23] J. W. Toigo. Avoiding a data crunch. *Scientific American*, 279(5):58–74, May 2000.

[24] Mustafa Uysal, Arif Merchant, and Guillermo Alvarez. Using MEMS-based storage in disk arrays. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003. To appear.

[25] P. Vettiger, M. Despont, U. Drechsler, U. Drig, W. Hberle, M. I. Lutwyche, H. E. Rothuizen, R. Stutz, R. Widmer, and G. K. Binnig. The "Millipede"—more than one thousand tips for future AFM data storage. *IBM Journal of Research and Development*, 44(3):323, 1999.

[26] John Wilkes. The Pantheon storage-system simulator. Technical Report HPL–SSP–95–14, Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, May 1996. http://www.hpl.hp.com/SSP/papers/PantheonOverview.pdf.





Optimal Configuration	cello99	cello92	tpcd	snake
$\delta_x(\mu m)$	9	9	5	9
$\delta_y(\mu m)$	22	22	40	22
T_{active}	2560	2560	2560	2560
Service time				
simulated (ms)	0.52	0.51	0.26	0.38
predicted (ms)	0.50	0.52	0.22	0.42
error	3.8%	1.9%	15.4%	13.5%
Default Configuration	**cello99**	**cello92**	**tpcd**	**snake**
Service time				
simulated (ms)	0.98	0.91	1.31	0.79
predicted (ms)	0.97	1.00	1.38	0.85
Difference from optimal				
simulated	88%	78%	403%	108%
predicted	94%	92%	527%	102%

Figure 7: Performance of configuration minimizing service time when capacity and request sizes are fixed. Above: Graph of the service time for a range of values for the movement range in Y for the simulated trace workloads. The movement range in X was adjusted at each point to keep the capacity at 2GB. Below: Experimentally obtained and predicted service times for the four workloads, and optimal configurations.

Optimal Configuration	cello99	cello92	tpcd	snake
$\delta_x(\mu m)$	5	5	5	5
$\delta_y(\mu m)$	29	27	80	32
T_{active}	320	320	320	320
Service time				
simulated (ms)	0.80	0.83	1.08	0.70
predicted (ms)	0.77	0.78	1.06	0.69
error	3.8%	6.0%	1.9%	1.4%
Default Configuration	**cello99**	**cello92**	**tpcd**	**snake**
Service time				
simulated (ms)	0.98	0.91	1.31	0.79
predicted (ms)	0.97	1.00	1.38	0.85
Difference from optimal				
simulated	23%	9.6%	21.3%	13%
predicted	26%	28.2%	30.2%	23%

Figure 8: Performance of configuration minimizing service time for a fixed number of active tips. Above: Graph of the service time for a range of values for the movement range in Y for the simulated trace workloads. Below: Experimentally obtained and predicted service times for the four workloads and optimal configurations.

Varied parameter	*Workload*							
	cello99		cello92		snake		tpcd	
	mean error	RMS error	mean error	RMS error	mean error	RMS error	mean error	RMS error
δ_x	2.2 %	0.03 ms	8.0 %	0.10 ms	7.8 %	0.09 ms	1.4 %	0.02 ms
δ_y	0.8 %	0.01 ms	6.2 %	0.07 ms	8.3 %	0.08 ms	1.4 %	0.02 ms
A	1.1 %	0.02 ms	6.0 %	0.07 ms	8.4 %	0.08 ms	3.2 %	0.03 ms
T_y	1.6 %	0.02 ms	6.6 %	0.07 ms	6.7 %	0.06 ms	1.4 %	0.02 ms

Table 3: Comparison of the service time of several simulated workloads to our model as both a percentage difference and a root mean squared difference, as we vary several parameters.

parameter space to find a set of values that optimized the given metric. We varied each parameter by intervals of 1μm for the movement range in both X and Y and 10 for the number of active tips.

5.1 Minimizing Service Time for a Fixed Capacity and Request Size

For a given bit density, the capacity of a MEMS device is determined by the product of active tips and movement ranges in X and Y. We looked for a configuration that minimized mean response time for a probe-based storage device with a fixed capacity of 2GB.

Given this constraint, our model identified an optimal configuration for each workload as shown in Figure 7b. We can see that the optimum number of active tips is at the top of the range for the values of that parameter. This is not surprising, because additional active tips lower service time. The movement ranges in X and Y are small, which also reduces service time. If we compare the values of the optimal configurations to the behavior of the experimentally obtained values for the service time in Figure 7a we can see that our predictions point out very well where the minimum of the service time will occur.

Note that the error is greatest for the tpcd workload, where the model generally underestimates the service time. Specifically, the model underestimates the seek time for tpcd because the distribution of seeks in the real workload does not fit our assumption of a uniformly random distribution. The tpcd workload has a significant number of large seeks that increase the overall average. As we change the movement range in X and Y, the model prediction changes more quickly than the real workload. Thus, at higher values of δ_y, the simulated and predicted service times converge.

5.2 Minimizing Service Time for a Fixed Number of Active Tips

In reality, design of a probe-based storage device is likely to be constrained by cost. We approximate this by fixing the number of active tips at 320 (assuming that the number of tips is proportional to the cost of the device) and determining a configuration that minimizes service time.

Figure 8b shows the configurations that minimize the service time for each workload. Figure 8a shows the service time when the movement range in Y is varied. Our model again does an accurate prediction of where the minimum service time occurs for all four workloads. In the case of snake, cello99 and cello92 this is just around 30μm, while in the case of tpcd this happens at the upper bound for the movement range in Y, 80μm. This makes sense because the tpcd workload has larger request sizes and runlengths, and therefore can benefit from the fewer turnarounds offered by long movements in Y.

6 Conclusions

The design space of probe-based storage devices is vast, but can be explored more quickly with the use of a parameterized model. We have presented a model for a probe-based storage device that allows a system designer to predict performance for a set of configurable parameters. We validated this model using a simulator for probe-based storage and several workloads and found that the error was within 15%. We successfully used this model to quickly identify optimal configurations to satisfy different performance objectives.

Acknowledgments

We are grateful to John Wilkes and Chuck Morehouse at HP Labs for important discussions and the use of the Pantheon simulator and traces. Thanks also to the anonymous FAST reviewers whose comments, together with shepherding by John Wilkes, greatly improved the draft. This research is based upon work supported by the National Science Foundation under grants NSF CCR-0073509 and 0093051.

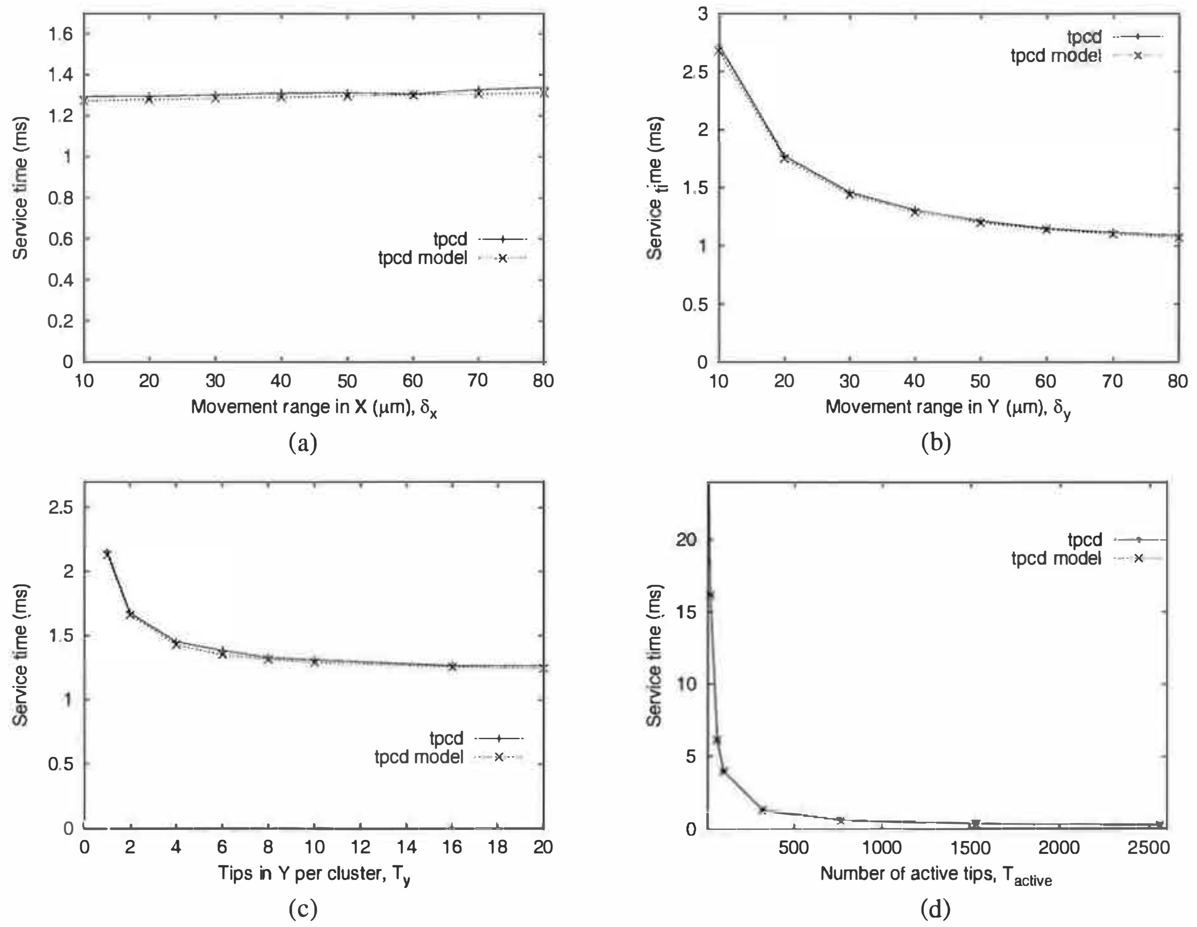


Figure 6: Graphs of the service time of the simulated tpcd workload and values calculated using our probe-based storage model. The initial configuration used had a 40μm movement range in both X and Y, 320 active tips and 10 tips in Y per cluster. (a) varying the movement range in X. (b) varying the movement range in Y. (c) varying the number of active tips. (d) varying the number of tips in Y per cluster.

Figures 5c and 6c confirm our expectation that the number of tips in Y per cluster has a negligible effect on performance at values greater than two.

Finally, Figures 5d and 6d show that increasing the number of active tips increases the level of parallelism inside the device, decreasing transfer time. The smaller transfer time at higher numbers of active tips makes errors in seek time more pronounced, causing the deviation between models and simulation.

We show the model error for each configurable parameter in Table 3. The percent error is calculated by computing the average percentage difference of the model-predicted values from the simulated values at each data point. The RMS difference is the root-mean squared vertical difference between the model predictions and the simulated workload value (in ms). The data in Table 3 show that our model closely approximates the simulated service time for all four workloads. Even though we mapped workloads to devices with different capacities and assumed random distribution of requests, our model was within 8% of the simulated values.

5 Optimal System Design

We used our probe-based storage model to optimize design of MEMS devices for specific performance goals. We used the range of values for each configurable parameter shown in Table 1 to bound our search, with the exception of the number of tips in Y per cluster. Because this parameter has a negligible contribution to the transfer time when its value is greater than 2, as can be seen from Figures 5b and 6b, we set it to the default value 10.

To find the optimal set of configurable parameters, we exhaustively searched the entire range of the configurable

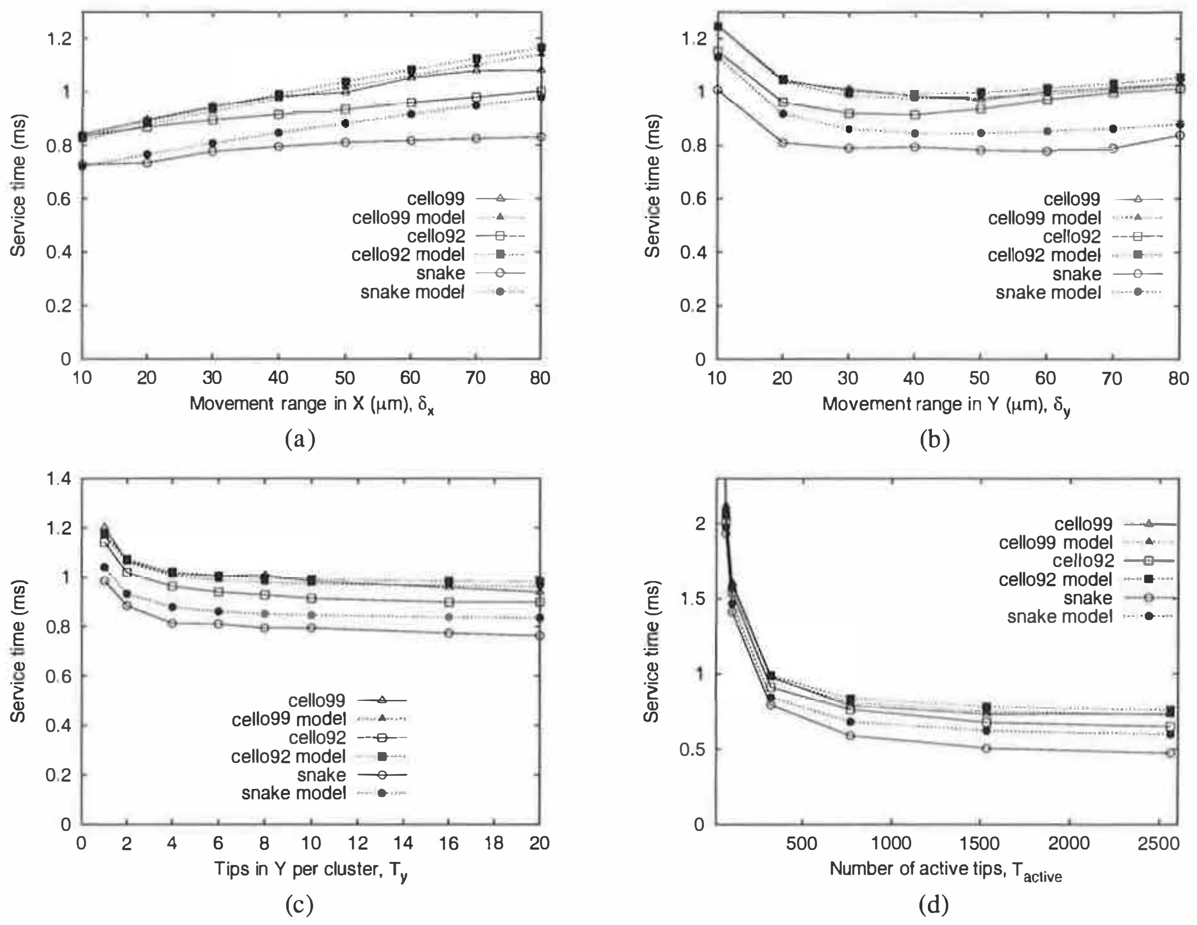


Figure 5: Graphs of service time of the simulated snake, cello92 and cello99 workloads and values calculated with our probe-based storage model. The initial configuration used had a 40μm movement range in both X and Y, 320 active tips and 10 tips in Y per cluster. (a) varying the movement range in X; (b) varying the movement range in Y; (c) varying the number of active tips; (d) varying the number of tips in Y per cluster.

around default values for which behavior is well understood [22].

Figures 5a–d show the results we obtained for snake, cello92, and cello99 workloads, and 6a–d show our results for the tpcd workloads. In Figure 5a, we see that for snake and cello92, as δ_x increases, the predicted service time deviates more from the experimental values. This is because the original disk from which the traces are taken is smaller than the probe-storage devices we are mapping those requests to. Thus, seeks occur over only a small fraction of the movement range in X, and our model overestimates the seek time as the device capacity increases. For cello99 and tpcd, traces taken from disks with higher capacity, this effect is much smaller. In fact, the error decreases in the case of the tpcd workload, as we can see from Figure 6a.

In general, model error decreases as we increase movement range in Y (Figure 5b). Because the sectors are ordered vertically, seeks are distributed across the entire Y movement range during the workload simulation. Thus the seek time predicted by our model better approximates the experimentally obtained values. This error becomes even smaller for higher movement ranges in Y, when seeks in Y dominate those in X. We can see from Figure 5b that when the movement range in Y is greater than 20μm, service time for cello92, snake and cello99, which have similar request sizes and runlengths, does not change significantly. Increasing the movement range in Y increases the average number of bytes that can be read before the mover must turnaround, however, these workloads with small requests do not benefit from this. In contrast, tpcd has a much larger request size and runlength, hence its service time is minimized for greater values of the movement range in Y, as shown in Figure 6b.

parameters, we obtained $\alpha = 75\sqrt{5}/(8\sqrt{\pi\delta})$, or $\alpha = 11.83/\sqrt{\delta}$.

The average value of the seek time in one direction, taken over all the requests for which its seek time was greater than that in the other direction, can be estimated by integrating its probability function over all possible times (0 to t_{max}), shown in Equation 15.

$$t_{average\ seek}(\delta) = \int_0^{t_{max}} P(t)tdt \quad (15)$$

When we substitute in Equations 12, 13 and 14 for $P(t)$ we obtain Equation 16:

$$t_{average\ seek}(\delta) = \frac{1}{8}\sqrt{\frac{\pi\delta}{5}} \quad (16)$$

Seeking in X involves a settling time t_{settle} that we add to the prediction for the average seek time in X. The final formulae for seek times in X and Y are shown in Equations 17 and 18.

$$t_{seek\ x}(\delta_x) = \frac{1}{8}\sqrt{\frac{\pi\delta_x}{5}} + t_{settle} \quad (17)$$

$$t_{seek\ y}(\delta_y) = \frac{1}{8}\sqrt{\frac{\pi\delta_y}{5}}, \quad (18)$$

We can now substitute Equations 17 and 18 in Equation 9 to obtain an expression for the seek time, which is shown in Equation 19.

$$t_{seek}(\delta_x, \delta_y) = \frac{1}{\sqrt{\delta_x} + \sqrt{\delta_y} + 8t_{settle}\sqrt{\frac{5}{\pi}}} \left((\sqrt{\delta_x} + 8t_{settle}\sqrt{\frac{5}{\pi}})(\frac{1}{8}\sqrt{\frac{\pi\delta_x}{5}} + t_{settle}) + \delta_y \frac{1}{8}\sqrt{\frac{\pi}{5}} \right) \quad (19)$$

4.2 Service Time Model

In Section 4.1.2 and Section 4.1.1 we obtained expressions for the seek and transfer times. We can now substitute these equations (19 and 8) for t_{seek} and $t_{transfer}$ in Equation 2. This gives us an expression for the service time as shown in Equation 20:

Workload	*Average request size*	*Runlength*
cello92	6.4 KB	6.5 KB
snake	6.8 KB	8.9 KB
cello99	6.8 KB	7.2 KB
tpcd	29.3 KB	252.0 KB

Table 2: Average request size and runlength values for workloads used in our simulations.

$$t_{service}(\delta_x, \delta_y, T_{active}, T_y, r, r_l) = \frac{r}{r_l(\sqrt{\delta_x} + \sqrt{\delta_y} + 8t_{settle}\sqrt{\frac{5}{\pi}})} \left((\sqrt{\delta_x} + 8t_{settle}\sqrt{\frac{5}{\pi}})(\frac{1}{8}\sqrt{\frac{\pi\delta_x}{5}} + t_{settle}) + \delta_y \frac{1}{8}\sqrt{\frac{\pi}{5}} \right) + \frac{8rd_b}{T_{active}}(\frac{1}{\delta_y}(t_{TA} + \frac{t_{XM}}{T_y}) + \frac{1}{v_0}) \quad (20)$$

4.3 Model and Simulation Comparisons

To compare our model with the performance of a prototypical probe-based storage device we conducted experiments using the Pantheon simulator from HP. We used a probe-based storage device simulator that implements the architecture and layout described in Section 3 [22], using a variant of the unconstrained sled model [10, 22]. We used four workloads: 1992 cello (4% sequential, /news partition, most requests smaller than 8KB, sector size is 256 bytes) [15], 1992 snake (23% sequential, /usr2 partition, most requests smaller than 8KB, sector size is 512 bytes) [15], 1999 cello (disk 128, 30% sequential, large requests where more than half are larger than 8KB, sector size is 1024 bytes) [22] and 1999 tpcd (disk 80, which accounted for 12.5% of the requests). All traces recorded one week of activity. In our simulations, we issue the requests at the original times they occur in the traces, so queuing behavior in the original workloads will not be as pronounced on the faster probe-based storage devices.

All original traces were recorded by HP Laboratories. The average request size and runlength for each workload are shown in Table 2.

We used as default physical parameter values from Table 1 and varied configurable parameters one at a time within the ranges shown in same table. These ranges were selected because they represent reasonable variation by the product of the number of bits in Y times the number of tips in Y per cluster (T_y), as shown in Equation 7.

$$n_{XM} = \frac{r_f}{n_{bitsinY} \times T_y}. \quad (7)$$

The number of bits in Y is equal to the movement range in Y divided by the bit-width.

Substituting Equations 4–6 in Equation 3 gives us an expression for the transfer time in milliseconds shown in Equation 8.

$$t_{transfer}(\delta_y, T_{active}, T_y, r) = \frac{8rd_b}{T_{active}}\left(\frac{1}{\delta_y}(t_{TA} + \frac{t_{XM}}{T_y}) + \frac{1}{v_0}\right) \quad (8)$$

4.1.2 Seek Time Model

Seek time is the time that it takes to reposition the mover above the relevant tips when servicing a new non-sequential request. To model the seek time we used a probabilistic approach. We assume that the starting locations of requests are uniformly distributed across the device.

Because the movement in X and Y may be considered independently [10, 6], the seek time can be computed as the greater of the seek times in X and Y. We computed the average seek time when the seek in X is greater ($t_{seek\ x}(\delta_x)$), and the average seek time when the seek in Y is greater ($t_{seek\ y}(\delta_y)$) and estimated these probabilities a and b, which add up to 1, to obtain Equation 9. We reason that a higher average seek time in one direction will have a higher impact on the overall seek time, so $a/b = t_{seek\ x}(\delta_x)/t_{seek\ y}(\delta_y)$.

$$t_{seek}(\delta_x, \delta_y) = a \times t_{seek\ x}(\delta_x) + b \times t_{seek\ y}(\delta_y) \quad (9)$$

While several models exist to estimate the seek time, the one we adopted is based on simple acceleration rules from Newtonian mechanics [13, 10] and is given in Equation 10, for a specified distance δ. The mover can seek achieve a much higher velocity than is used to read/write because it can accelerate during the seek. At the end of a seek, the data must be accessed by moving in the Y direction. Therefore, a settle time, t_{settle} (for the mover to position itself accurately) applies only to the calculated seek time in X, since the mover has to come to a complete stop in that direction. Notice that seeking takes place within one tip area. Therefore, the seeking distance and the seek time are relatively short, and depend on the mover movement range.

$$t(\delta) = \sqrt{\frac{2\pi\delta}{a_0}} \quad (10)$$

We calculate the average seek time in X and Y as follows. First, we calculate the probability function $p(d)$ of an incoming request incurring a certain movement d over the tip array either X or Y (we can consider each direction independently). Because the starting sectors are uniformly distributed, the distances in each direction will be the distance between two uniformly distributed starting sectors. This probability will vary linearly with distance and will be at its maximum for a zero displacement, and it will be equal to zero when the displacement is equal to the movement range. The form of the equation is $p(d) = c - bd$, where c and b are constants. Because p is a probability function, the area under it in the range 0 to δ, where δ is the movement range in X or Y, which is $c\delta/2$ will be equal to 1. Additionally, when $d = \delta$, the probability is equal to zero, $p(\delta) = 0$. Using these constraints, we can calculate the values for the constants c and b, which we substitute back in the expression for $p(d)$ to obtain Equation 11:

$$p(d) = 2(\delta - d)/\delta^2 \quad (11)$$

Unfortunately, we need to calculate seek time distributions, not distance distributions. We can use Equation 10 to express t as a function of a displacement d, converting Equation 11 from the distance domain to the time domain. This gives us the probability of a seek incurring time t in either X or Y. After substituting 250 m/s^2 for the physical parameter acceleration, a_0, we obtain Equation 12:

$$p(t) = \frac{500t}{\pi\delta}(1 - \frac{250t^2}{2\pi\delta}) \quad (12)$$

The movement ranges δ_x and δ_y in X and Y from Table 1 specify the maximum distance we can move in each direction. We use them in Equation 10 to find the maximum seek time t_{max} in X or Y, shown in Equation 13.

$$t_{max} = \sqrt{\frac{2\pi\delta}{250}} \quad (13)$$

The actual seek time is the greater of the seek times in X and Y, so the probability distribution function $P(t)$ of the seek time when it is greater in one direction than the other will be different than $p(t)$. Reasoning that taking the maximum will bias us towards larger seek times, we approximated $P(t)$ with Equation 14.

$$P(t) = \alpha p(t)t \quad (14)$$

To solve for the constant factor α we integrated $P(t)$ from 0 to t_{max} and normalized it (because it is a probability function). Using our default values for physical

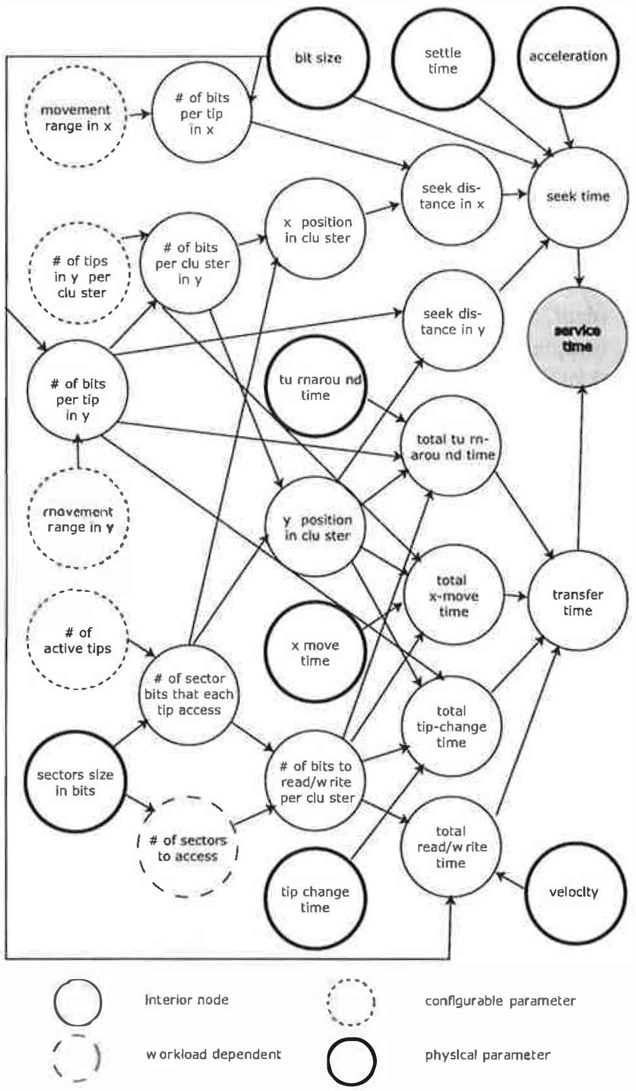


Figure 4: A simplified design space parameter dependency graph. The head of the graph represents the service time, which is the minimization target. From the graph we can identify the parameters (represented by the leaves) on which service time depends.

or write the data by moving the mover over the tip array with a constant velocity (*read/write* time, or t_{RW}). Second, if a request spans more than one bit column, we must add the time that it takes for the mover to reverse direction (e.g., from moving down to moving up); we call this time the *turnaround* time (t_{TA}). To minimize wasted space, a sector can begin at any bit within a column, and continue at the next column. The MEMS device controller is responsible for tracking the starting positions of sectors. Thus, a single read/write request may require one or more bit column movements in the X direction. This third component is called an *x-move* (t_{XM}). The last component is the time that it takes to switch between sets of active tips, or *tip switch* time (t_{TS}).

Thus, the transfer time is a sum of four terms: n_{TA}, the number of turnarounds, multiplied by t_{TA}, the turnaround time; n_{TS}, the number of tip switches times t_{TS}, the tip switching time; n_{XM}, the number of moves in X, times t_{XM}, the time to move one bit in X, and t_{RW}, the time it takes to actually read the data. Equation 3 shows this combination.

$$t_{transfer} = n_{TA} \times t_{TA} + n_{TS} \times t_{TS} + n_{XM} \times t_{XM} + t_{RW} \quad (3)$$

As described in Section 3.2,the data to be read or written is divided among all active tips, which work in parallel. Each tip reads r_f bits, which is equal to the number of bits per request ($8\ r$) divided by the number of active tips T_{active}, as shown in Equation 4.

$$r_f = \frac{8r}{T_{active}} \quad (4)$$

The read/write time, t_{RW}, depends on the number of bits r_f that each active tip has to read, the velocity of the device (v_0), and the bit width d_b (to translate the number of bits into distance). To read or write each bit, it will take d_b/v_0 time for each active tip to move over it. This relationship is given in Equation 5.

$$t_{RW} = \frac{r_f \times d_b}{v_0} \quad (5)$$

The number of turnarounds, X-moves, and tip changes depend on the number of requested bits that each active tip accesses, the starting position, and the the number of bits in a bit column ($n_{bitsinY}$). To calculate this quantity we divide the movement range in Y by the bit-width. Using the layout described in Section 3.2, the number of turnarounds is the same as the number of tip switches, because a tip switch is necessary at every turnaround. The X-move component also depends on the number of tips in Y per cluster that determines the cluster dimension (i.e., the number of bits in one cluster column).

Specifically, the average number of turnarounds (or tip switches) per request is the ratio of the number of bits each tip must access to service the average size request (r_f) and the number of bits in each column ($n_{bitsinY}$), as shown in Equation 6.

$$n_{TA} = \frac{r_f}{n_{bitsinY}} \quad (6)$$

The average number of X-moves is the ratio of the average requested number of bits per active tip (r_f) divided

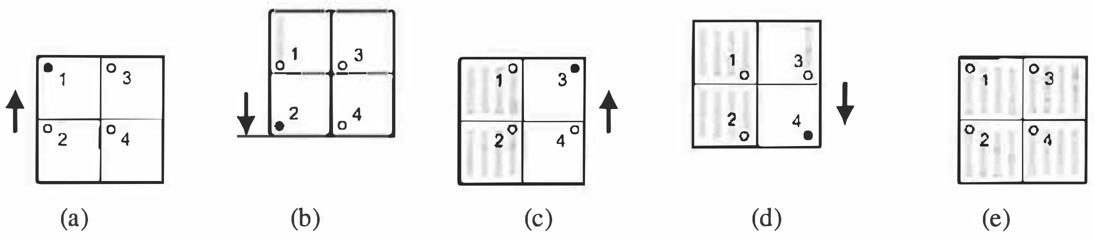


Figure 3: Data access for a single mover with four clusters and one tip per cluster. (a) Tip 1 is activated and the mover slides up, reading the column of bits the tip passes over. (b) Tip 2 is activated after the shaded bits have been read, and the mover reverses direction. (c) After all columns have been read, with an x-move between each column to reposition the tips over the new column of bits, tip 3 is activated and the mover slides up, analogously to (a). (d) Tip 4 is activated and the mover reverses direction, analogously to (b). (e) All bits in the mover are read.

is striped among them all. First the stream is split into groups of four bits each, as depicted by the different gray shades. Then each such group of four bits is interleaved between the four active tips, as shown by the arrows in Figure 2.

4 Service Time Model

The performance of a particular probe-based storage device depends on its configuration and the workload that it is subject to. We can measure performance using a model of probe-based storage implemented in Pantheon, an I/O device simulator created by Hewlett-Packard [26, 16]. We used this model to explore the behavior of probe-based storage under several workloads and identify a configuration with performance characteristics similar to disks [21]. However, the number of different configuration parameters is too large to use such a simulator to explore the design space exhaustively.

To address this problem, we created an analytical model that predicts the response time for various configurations. We make the simplifying assumption that requests are uniformly distributed across the device. This assumption does not hurt us as significantly as it would with disk drives because seek time is dominated by transfer time except at large degrees of parallelism [22]. Furthermore, as shown in [21], seek time is not as sensitive to other architectural parameters as transfer time.

4.1 Performance Dependencies

Based on the architecture and the layout model described in Section 3, we know that the design space of probe-based storage is relatively complex and involves many parameters such as the number of active tips per device, mover's movement range, and acceleration. To choose one configuration over another, we must understand how the physical configuration affects the performance. Therefore, we analyze the dependencies in the model between these parameters and the service time, which we wish to minimize.

Figure 4 is a dependency graph showing how performance is related to probe-based storage parameters [21]. The graph edges represent the dependencies, the leaf nodes are parameters, and the remaining nodes are intermediate variables. The dependencies in this graph reflect the parameters and layout presented in Section 3, although other models could be used both for the layout and for the physical behavior of the device [10]. The target in the graph is the service time, which is the sum of seek time and transfer time, as shown in Equation 1. We model each of these components separately:

$$t_{service} = t_{seek} + t_{transfer} \tag{1}$$

Our model for seek time, described in Section 4.1.2, assumes uniformly distributed requests. To adjust for sequentiality in workloads, we parameterize Equation 1 with the runlength calculated from the trace. To compute the runlength, we calculate the length, in bytes, of each sequential run in the trace and average these runs to compute the runlength in bytes, r_l.

Only the first request of a sequential run will require repositioning the mover, incurring seek time. Thus the ratio of average request size (r) to runlength (r_l) represents the fraction of requests that require seeks. We modified the service time equation (1) to use the runlength and obtained Equation 2:

$$t_{service} = \frac{r}{r_l} t_{seek} + t_{transfer} \tag{2}$$

4.1.1 Transfer Time Model

Transfer time ($t_{transfer}$), the time to actually read/write the data, has four components. The first is the time to read

Configurable parameter	Default value	Symbol
Movement range		
in X (5–80μm)	40μm	δ_x
in Y (5–80μm)	40μm	δ_y
Active tips (10–2560)	320	T_{active}
Tips in Y		
per cluster(1-20)	10	T_y
Physical parameter		
Settle time	200μs	t_{settle}
X-move time	1ms	t_{XM}
Velocity	0.05m/s	v_0
Turn around time	400μs	t_{TA}
Tip change time	0	t_{TS}
Bit width	50nm	d_b
Acceleration	250m/s^2	a_0
Active tips per cluster	1	$T_{cluster}$
Workload parameter		
Average request size	calculated from workload	r
Runlength	calculated from workload	r_l

Table 1: Configurable and physical probe-based storage architecture parameters.

Figure 1b magnifies the mover area from Figure 1a. This shows that a mover is divided into one or more *clusters*. Each cluster can read data independently of the others, which provides higher parallelism to the device. As an example, the mover in Figure 1b is subdivided into four clusters. Each cluster is a media area that is accessed by many tips, only one of which can be active at a time. Using several tips in parallel, one from each cluster, compensates for the low data rate of each individual tip, which is on the order of 1Mbit/sec. The number of bits accessed simultaneously is equivalent to the number of clusters per mover times the number of movers in the device. We call this the number of *active tips*.

The mover's range of movement and the bit size determine the amount of data that can be manipulated by one tip, or *tip area*. Because several tips are active at a time, different tip areas of the mover are manipulated simultaneously, as depicted by the shaded rectangles in Figure 1b. Different areas of the mover are accessed by switching between sets of active tips.

Many architectural configurations are possible for probe-based storage. For example, we might vary the number of active tips, the mover's movement range, the media density, and so on. The values summarized in Table 1 are reasonable defaults for probe-based storage devices [25, 6, 1], and are the parameters we used in simulations unless otherwise specified.

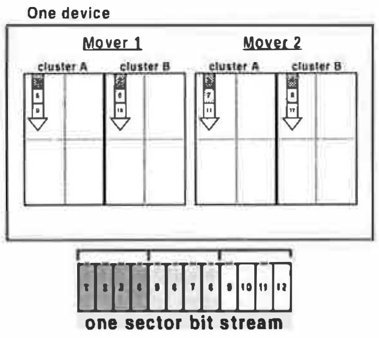


Figure 2: Example data layout over two movers in one MEMS device.

3.2 Data Layout

Traditional disk data layout minimizes seek time and rotational latency. Analogously, we chose a layout for probe-based storage that has a similar sequential ordering to minimize movement for consecutive requests accessing sequential data.

Each mover is divided into several clusters, each of which is an area covered by a grid of tips. For each cluster, only one read/write tip can be active at a time. This cluster design helps to minimize power consumption while increasing the size of the swept area. Given this parallel architecture, where several tips can read/write data in parallel, it was reasonable to stripe the data between the clusters. We chose to work with bit-level striping because it maximizes the throughput when there is only one outstanding request.

Figure 2 shows the layout of a MEMS device with two movers, two clusters per mover and four tips per cluster. This makes the tip area a quarter of the cluster's area. One tip per cluster may be active at a time; therefore, in this device, four tips are active at a time. Consequently, a sector of data is read/written by the four active tips simultaneously. The sector is bit-interleaved between the two clusters, as depicted by the pattern of the bitstream in Figure 2. The data is accessed while the mover is moving in the $\pm y$ direction. Figure 3 shows how data is accessed on a single cluster.

A single device may contain several movers, increasing parallelism. To exploit this secondary level of parallelism, we stripe data bitwise among all the clusters of these movers. For example in Figure 2 there are four simultaneously active tips, one per cluster, and the bitstream tion Processing and Storage Systems (CHI^{2}PS^2 [3]), Despont *et al.* [25], Mamin *et al.* [11], and the Atomic Resolution Storage (ARS) project at Hewlett-Packard Laboratories [23]. While these projects use different recording technologies, they are all based on tip arrays and media sleds. Thus, the models we describe can be tuned with different input parameters to describe these systems.

Disk modeling has been traditionally used to design storage systems that use hard drives, and a complete survey of this work is beyond the scope of this paper. Because disk performance is so workload-dependent, most simplifying assumptions cause large errors [16]. Shriver [19] developed some analytic models to incorporate effects of disk caching and I/O workload variation.

Because probe-based storage devices do not yet exist, it is particularly useful to create models of them that can yield insights into performance. Yang and Madhyastha created several physical models for seek time of probe-based storage, defining a performance range for a specific hardware configuration [10]. A different model was presented by Schlosser and Griffin *et al.* [6, 17], who conclude that a probe-based storage device can improve applications performance by a factor of three over disks. They compare and contrast probe-based storage and traditional disk drives, and study how aspects of the operating system need to be changed when a system is built with probe-based storage [7]. Uysal *et al* [24] evaluate several hybrid MEMS/disk architectures, showing that hybrid architectures can give performance benefits similar to replacing disks with probe-storage devices (at lower price-performance). Ying *et al* [9] used this seek-time model to devise policies for power conservation. However, these studies all rely upon trace-driven simulation of traces. In contrast, our focus here is to develop an analytical model for a wide range of probe-based storage characteristics that can be used for such performance research.

Probe-based storage devices are much faster than traditional disk drives, making the question of how probe-based storage may be integrated into the memory hierarchy very important. Griffin *et al* [18] show that using probe-based storage as a disk replacement will improve overall application runtime by a factor of 1.9–4.4, and when used as a disk cache can improve I/O response time by up to 3.5 times.

3 Probe-Based Storage

Probe-based storage devices have a wide range of configurable parameters. In Section 3.1 we describe how a probe-based storage device works and highlight the parameter space. In Section 3.2 we show how data may be stored on such a device.

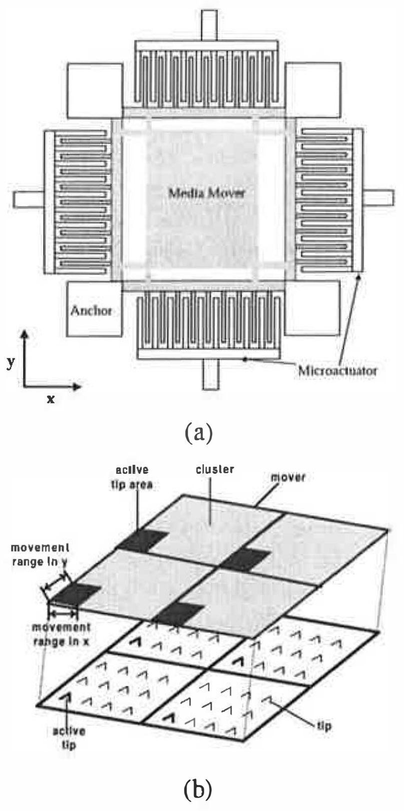


Figure 1: (a) Mover and microactuators. The device is shown from above; the tip array lies below the mover surface. (b) A probe-based storage device that includes one mover (the shaded areas) above a tip array. The mover is divided into four clusters, each of which has 12 dedicated tips. One tip per cluster is active at a time (the shaded tips), accessing the tip areas depicted by the dark rectangles. The mover has limited range of movement over the tip array in the X and Y directions.

3.1 Architecture

Figure la is a top view of a typical probe-based storage device. In this figure, the shaded parts move and the unshaded parts are stationary. The media *mover* is suspended above a surface on which a grid of many probe *tips* are embedded. Collectively, the tip array is the logical equivalent of the read/write heads of a traditional disk drive. Voltage applied to the fingers of the microactuator combs exerts electrostatic forces on the mover that cause it to move in the X and Y directions, overcoming the forces exerted by the anchors and beams that keep it in place. To service a read or write request, the mover first repositions itself so that the tip array can access the required data. This repositioning time is called *seek time.* The mover then accesses the data while moving at a constant velocity in the Y direction, incurring *transfer time.*

Optimizing Probe-Based Storage

Ivan Dramaliev
Department of Computer Science

Tara Madhyastha
Department of Computer Engineering

University of California Santa Cruz
1156 High Street, Santa Cruz, CA 95064
{ivan,tara}@soe.ucsc.edu

Abstract

Probe-based storage, also known as micro-electric mechanical systems (MEMS) storage, is a new technology that is emerging to bypass the fundamental limitations of disk drives. The design space of such devices is particularly interesting because we can architect these devices to different design points, each with different performance characteristics. This makes it more difficult to understand how to use probe-based storage in a system. Although researchers have modeled access times and simulated performance of workloads, such simulations are time-intensive and make it difficult to exhaustively search the parameter space for optimal configurations. To address this problem, we have created a parameterized analytical model that computes the average request latency of a probe-based storage device. Our error compared to a simulated device using real-world traces is small (less than 15% for service time). With this model we can identify configurations that will satisfy specific performance objectives, greatly narrowing the search space of configurations one must simulate.

1 Introduction

In the last 20 years microprocessor speeds have been improving by 50% to 100% per year [12] and memory capacities have been increasing by 60% per year [8]. Unfortunately, secondary storage has not kept pace. There are several limitations for today's disk technologies, such as disk rotational speed, bit density (which is limited by the superparamagnetic effect [23, 4]) and read-write head technology [23]. Academia and industry are developing new technologies to bypass these limitations. These technologies include holographic storage [14, 20, 23], atomic force microscopy (AFM) [11, 5, 23], and MEMS storage [3, 2, 6, 23]. Each of these storage alternatives has inherent tradeoffs that govern its use in a system. In this paper, we focus on the performance characteristics of one specific new technology: MEMS storage. MEMS storage technology is based on an array of atomically-sharp probe tips that read and write data on the storage medium. While several alternative styles of MEMS storage are currently being explored, all of these anticipated devices share certain common characteristics: they support high throughput, high parallelism and high density. Data is accessed in a disk on a rotating media platter, while in MEMS storage the media does not rotate but moves in a rectilinear fashion. The design space of MEMS storage is particularly interesting because we can architect these devices to different design points, each with different performance characteristics. This makes it more difficult to understand how to use probe-based storage in a system. Exhaustive simulation is at best extremely time-consuming and at worst impossible, depending on the search space. In our experiments it took approximately 20 minutes on an Intel Pentium 3 500MHz machine to run the simulation of a workload using a single configuration. The design space that were concerned with included over a million configurations, and it would take 14 days on a 1000 node cluster to test them all. To address this problem, we have created a parameterized analytical model that computes the average request latency of a MEMS storage device. Our error compared to a simulated device using real-world traces is small (within 15% error for service time). We used this model to identify optimal configurations given such constraints as capacity and throughput.

The remainder of this paper is organized as follows. In Section 2 we describe related work. We present architecture and design layout of a MEMS storage device in Section 3. In Section 4 we derive the equations governing the service time of a MEMS storage device under uniformly distributed accesses. We use this model to identify optimal configurations subject to user-specified constraints in Section 5. We conclude in Section 6.

2 Related Work

Current projects on probe-based storage include those by the Carnegie Mellon Center for High Integrated Informa-

(RAID). In Harran Boral and Per-Ake Larson, editors, *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–16, June 1988.

[17] D. Reinsel. Worldwide hard disk drive market forecast and analysis, 2000-2005. IDC, May 2001. IDC Report 24603.

[18] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[19] S. Savage and J. Wilkes. AFRAID – a frequently redundant array of independent disks. In *Proc. of the 1996 USENIX Technical Conference*, pages 27–39, January 1996.

[20] S.W. Schlosser, J.L. Griffin, D.F. Nagle, and G.R. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 1–12. ACM Press, November 2000.

[21] M. Sivan-Zimet and T. Madhyastha. Workload based optimization of probe-based storage. In *Proceedings of SIGMETRICS*, pages 256–7, June 2001.

[22] J.A. Solworth and C.U. Orji. Write-only disk caches. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 123–132, May 1990.

[23] J.A. Solworth and C.U. Orji. Distorted mirrors. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 10–17. IEEE Computer Society, December 1991.

[24] D. Stodolsky, G. Gibson, and M. Holland. Parity logging: Overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 64–75. IEEE Computer Society Press, May 1993.

[25] T.M. Wong and J. Wilkes. My cache or yours. In *Proc. of the 2002 USENIX Annual Technical Conference*, June 2002.

[26] J. Wilkes. The Pantheon storage-system simulator. Technical report, Storage Systems Program, Hewlett-Packard Laboratories, Palo Alto, CA, 29 December 1995.

[27] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 96–108. ACM Press, December 1995.

[28] J. Wilkes and R. Stata. Specifying data availability in multi-device file systems. *Operating Systems Review*, 25(1):56–9, January 1991.

Overall, we conclude that replacing disks with MEMS storage in disk arrays will improve performance and performance/cost, even if MEMS storage costs ten times as much as disks on a per-byte basis. Placing one copy of the data on MEMS storage is also effective, offering an intermediate cost and performance between conventional disk arrays and purely MEMS-based arrays.

Extensions of our work include studying the performance of the different alternatives after an unrepaired failure, in degraded mode and during online reconstruction. Another extension would be to incorporate reliability metrics into the architectural comparison when reliability estimates become available for MEMS-based storage devices.

Acknowledgements

Thanks are due to Miriam Sivan-Zimet for writing the initial version of the MEMS simulator module, Tara Madhyastha for making it available to us, Steve Schlosser for discussions on the subject of MEMS-based storage, Dick Henze and Bill Hooper for helping us find out the costs of the different technologies, and John Wilkes for helping us disentangle many Pantheon snares.

References

[1] G.A. Alvarez, W.A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 62–72. ACM Press, June 1997.

[2] D. Bitton and J. Gray. Disk shadowing. In Francois Bancilhon and David J. DeWitt, editors, *Proceedings of 14th International Conference on Very Large Data Bases (VLDB)*, pages 331–8. Morgan Kaufmann, August 1988.

[3] L.R. Carley, G.R. Ganger, and D. Nagle. MEMS-based integrated-circuit mass-storage systems. *Communications of the ACM*, 43(11):72–80, November 2000.

[4] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[5] T.C. Chiueh. Trail: a track-based logging disk architecture for zero-overhead writes. In *Proceedings of 1993 IEEE International Conference on Computer Design ICCD*, pages 339–343, October 1993.

[6] I. Dramaliev and T.M. Madhyastha. Optimizing probe-based storage. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, Mar-Apr 2003.

[7] R.M. English and A.A. Stepanov. Loge: a self-organizing storage device. In *Proceedings of USENIX Winter'92 Technical Conference*, pages 237–51. USENIX, January 1992.

[8] J.L Griffin, S.W Schlosser, G.R. Ganger, and D.F Nagle. Modeling and performance of MEMS-based storage devices. In *Proceedings of ACM SIGMETRICS*, pages 56–65, June 2000.

[9] T. Haining and D. Long. Management policies for non-volatile write caches. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference*, pages 321–328, February 1999.

[10] Hewlett-Packard Company, Palo Alto, CA. *OpenMail Technical Reference Guide*, 2.0 edition, 2001. Part No. B2280-90064.

[11] Y. Hu and Q. Yang. DCD - Disk Caching Disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pages 169–178. ACM Press, May 1996.

[12] Y. Hu, Q. Yang, and T. Nightingale. RAPID-cache - a reliable and inexpensive write cache for disk I/O systems. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 204–213. IEEE Computer Society, January 1999.

[13] A. Merchant and P.S Yu. Analytic modeling and comparisons of striping strategies for replicated disk arrays. *IEEE Transactions on Computers*, 44(3):419–433, March 1995.

[14] K. Mogi and M. Kitsuregawa. Hot mirroring: a method of hiding parity update penalty and degradation during rebuilds for RAID5. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 183–194, June 1996.

[15] C.U. Orji and J.A. Solworth. Doubly distorted mirrors. In *Proceedings of the 1993 ACM SIGMOD conference*, pages 307–316. ACM Press, May 1993.

[16] D.A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks

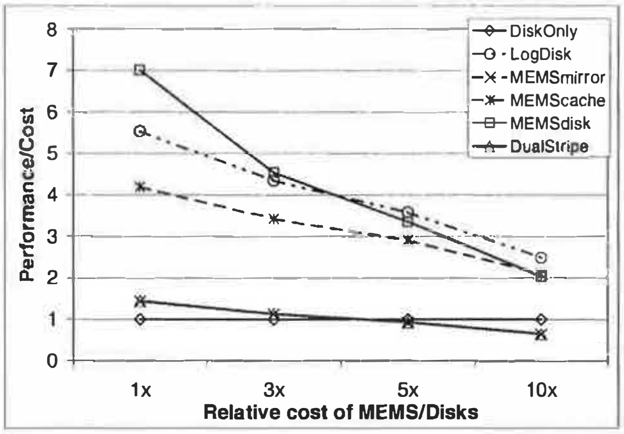


Figure 10: Performance/Cost of disk array architectures with MEMS-based storage.

Several papers have explored the use of logging (writing data to a sequential log) or eager-writing (writing to an unused location near the current position of the disk head). RAPID-Cache [12] provides redundancy to the NVRAM write cache through a logging disk, which is less expensive than replicating the NVRAM cache. In DCD (Disk Caching Disk)[11] and in Trail [5], a log disk is used to cache writes. Eager-writing was explored in the Loge disk controller [7]: writes were made to the free block closest to the current location of the disk head, and its location maintained through an indirection table in NVRAM. In Distorted Mirrors [23], data is mirrored, but only one of the copies (the master copy) is kept in a fixed order. Blocks in the master copy are updated in-place, but in the slave copy a block update can be written to the free block closest to the disk head. The main advantage is that write costs for the slave copy are lower than for mirrored disks where both copies are in a fixed location. In Doubly distorted mirrors [15], this is amended to defer the update to the master copy; the block is kept in a RAM cache, and redundancy is maintained by writing slave copies to both disks using eager-writing. The master location is updated from cache (and the slave location on that disk released) when there is a read from that cylinder, or the cache fills up, in which case the dirtiest cylinder is written out. Although this requires three disk writes for an update, the overall cost is lower than that of updating the master block immediately. Dual striping [13] attempts to reduce the cost of positioning time in a mirrored layout for reads while allowing load balancing across disks by using a large stripe size for one copy and a small stripe size for the other. Large reads can use the large stripe copy to reduce positioning time costs while small reads go to the small stripe copy.

The cost of small writes is particularly severe in RAID-4 and RAID-5 layouts, where every small write engenders four physical accesses: a read each of the old data and the corresponding parity (in order to compute the new parity value), and a write of the new data and the new parity. Parity logging [24] buffers parity updates in NVRAM and a log disk, eventually writing the parity out as large writes. AutoRAID [27] organizes the data hierarchically, with a RAID-10 level acting as a cache for a RAID-5 level; the RAID-5 level is log structured. Hot Mirroring [14] similarly combines RAID-1 and RAID-5 layouts, keeping hot data in the RAID-1 portion and cold data in the RAID-5 portion.

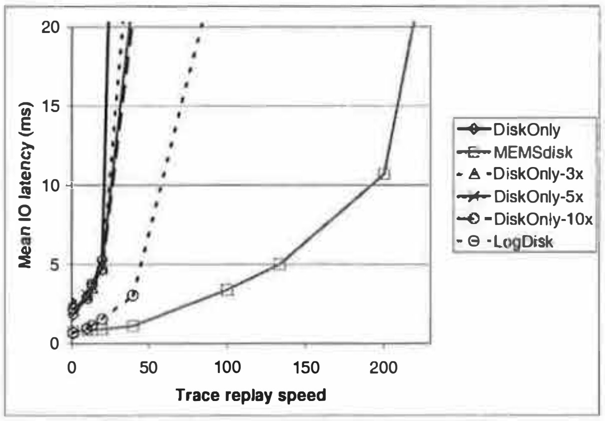
6 Conclusions and future work

We explored the performance and the performance/cost implications of incorporating MEMS-based storage into disk array architectures. We examined several possible placements for the MEMS storage in the disk array by (1) replacing all the disks with MEMS storage, (2) replacing the NVRAM cache with MEMS storage, and (3) replacing half the disks with MEMS storage. In the latter case, we proposed several novel alternative disk-array architectures designed to take advantage of the combination of disks and MEMS storage.

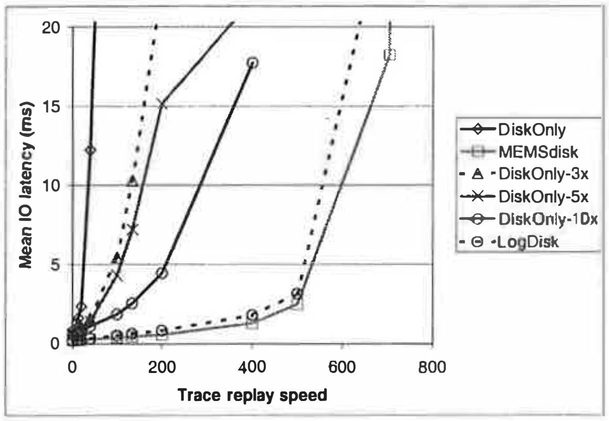
Replacing the disks with MEMS storage improves performance substantially in terms of latency (by a factor of 4 – 6.5) and throughput (by a factor of 4 – 28) depending on workload, but at high cost. Performance/cost, based on the average throughput of the trace workloads used, ranges between 2–7 times that of DiskOnly, depending on the MEMS/disk cost ratio.

The hybrid architectures, which store one copy of the data in MEMS storage, are able to achieve a significant fraction of the performance benefits of completely replacing disks with MEMS storage in disk arrays. Of the hybrid architectures studied, the LogDisk architecture offered the most consistent improvement in performance and the best performance/cost. The performance/cost of LogDisk is similar to that of purely MEMS-based arrays, and better than DiskOnly by a factor of 2.5–5.5, depending on the MEMS/disk cost ratio. Average latency is substantially lower than DiskOnly for all the hybrid architectures — by a factor of between 4 and 16 for the trace workloads studied here.

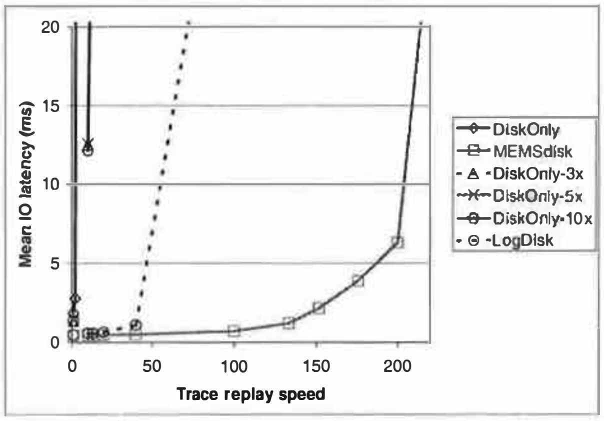
Replacing the NVRAM cache with a (much larger) MEMS cache is effective in reducing average response time by as much as 82%, but does not improve throughput because working set sizes are large. However, this may still be worth doing because of the low cost, as the performance/cost improvement over the conventional architectures ranged between 2.1–4.2, depending on the MEMS/disk cost ratio.



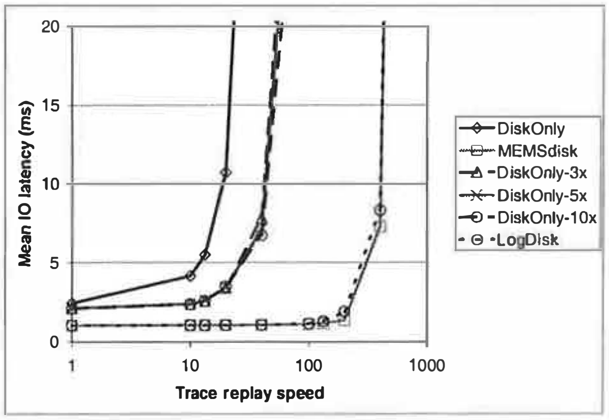
(a) *cello* trace



(b) *omail* trace



(c) *tpcc* trace



(d) *tpcd* trace

Figure 9: Comparison of architectures of equal cost for trace replays, in scenarios where the cost per byte for MEMS is 1 time, 3 times, 5 times, and 10 times that of magnetic disks.

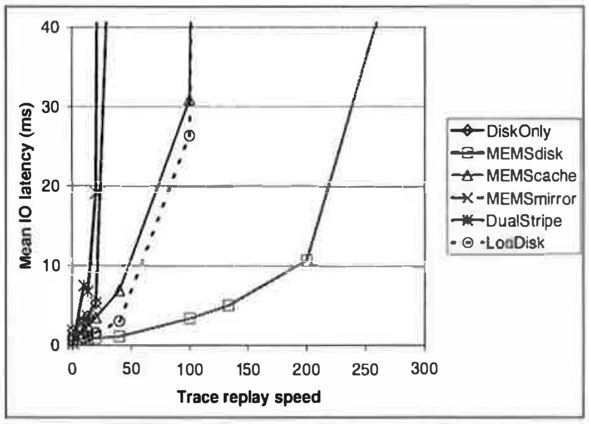
We conclude that it is more cost-effective to replace disks with MEMS storage than simply to add more disks.

Figure 10 compares the performance per unit cost of the hybrid architectures against MEMSdisk and DiskOnly. Performance is measured by the maximum throughput achieved, averaged across the four trace workloads. As expected, MEMSdisk is the most cost-effective architecture when the MEMS-based storage costs no more than disks. LogDisk and MEMScache have similar cost-performance, with LogDisk sligtly higher. The performance/cost of LogDisk declines more slowly than that of MEMSdisk as MEMS cost increases: when the MEMS/disk cost ratio is 10, its performance/cost exceeds that of MEMSdisk by 38%. The performance/cost of the remaining hybrid architectures (MEMSmirror and DualStripe) is about the same as that of the DiskOnly architectures; however, when the MEMS/disk cost ratio is 10, the performance/cost of these hybrids drops below that of DiskOnly.

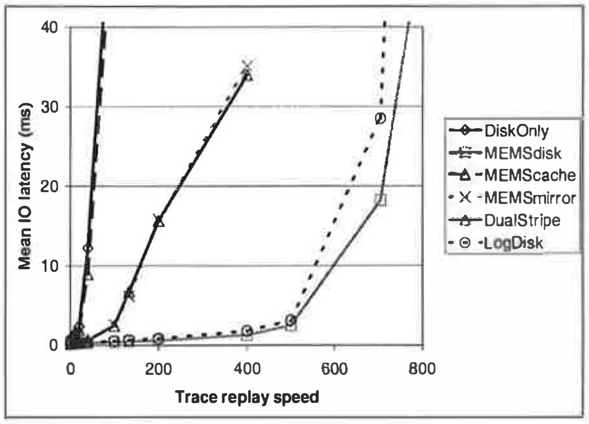
5 Related work

This paper combines the use of MEMS storage devices with several different redundancy schemes and layouts in efficient storage array architectures. The physical characteristics and performance of MEMS-based storage devices are discussed in several papers from the CMU Parallel Data Laboratory [3, 20, 8].

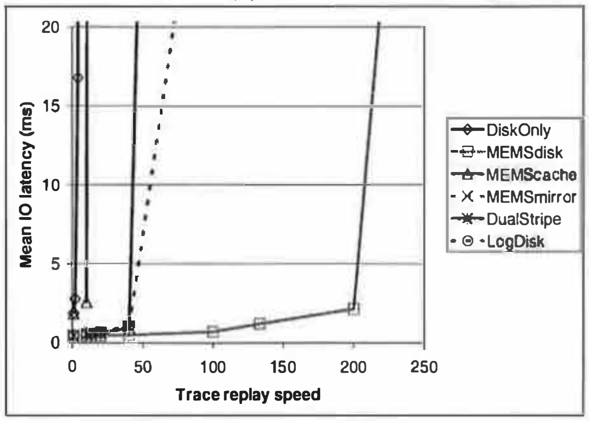
The use of redundant data layouts for reliability, load balance and improved performance is well established [1, 2, 16], and these are commonly used in modern disk arrays. In most such layouts, the performance of the disk is limited by the disk head seek time and rotational delays, particularly for workloads with small, non-sequential I/Os. Several mechanisms have been proposed to ameliorate the impact of positioning time for writes. A write cache can substantially reduce the number of disk writes and the perceived delay for writes [22, 9]; however, for reliability, these caches must generally use expensive NVRAM, ideally in a redundant configuration.



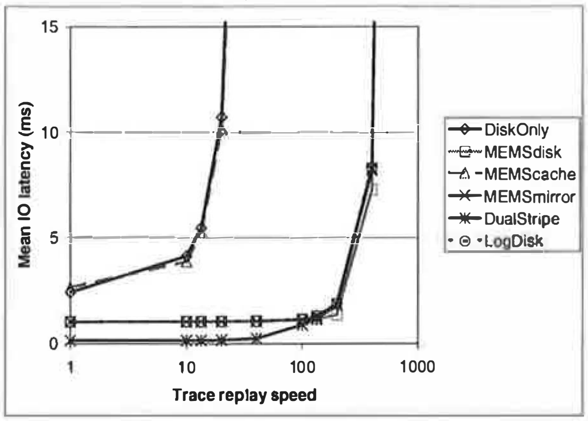
(a) *cello* trace



(b) *omail* trace



(c) *tpcc* trace



(d) *tpcd* trace

Figure 8: Comparison of architectures of the same capacity for the *cello*, *omail*, *tpcc*, and *tpcd* traces.

cache is able to sustain write bursts of much longer duration, there is a substantial performance penalty for each write compared to the NVRAM cache. On the other hand, the larger MEMS cache reduces the load on the back-end by eliminating most over-writes and coalescing dirty blocks in the cache.

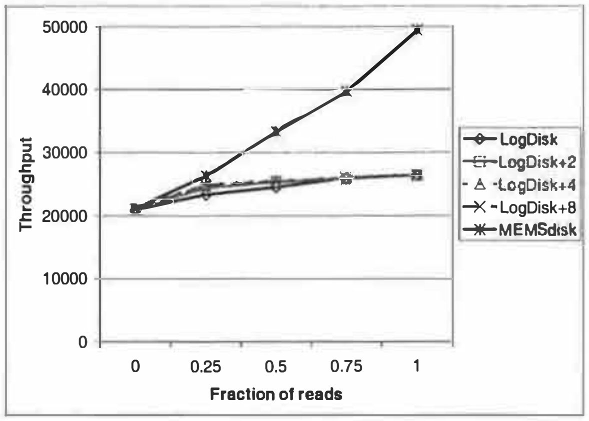
Overall, we conclude that replacing back-end disks in a disk array with MEMS storage has a dramatic impact on the performance of the array. A large part of that improvement can be obtained by placing only one of the two replicas of the data on MEMS storage, and it is effective to organize the disk replica in a log-structured fashion. Replacing the array NVRAM cache with a much larger MEMS cache is less effective in reducing the latency, and does not improve the throughput significantly.

4.3.4 Cost/performance analyses

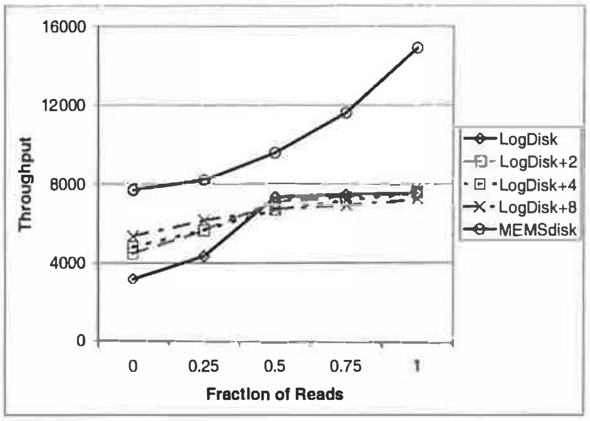
We now study cost/performance ratios in two different ways. First, we compare MEMSdisk with DiskOnly architectures of equal cost, *i.e.*, the "Isocost-X" architectures. Then, we compare the cost/performance of several hybrid architectures with DiskOnly and MEMSdisk.

Recall that the architecture cost is based on the cost of the disks, NVRAM and MEMS used. We used a cost of \$6/GB for disks, based approximately on 2001 list prices for enterprise-class disks [17] and \$8/MB for NVRAM, based on recent Dallas Semiconductor list prices. Since the cost for MEMS is unknown, we varied the relative per-byte cost of MEMS storage to disks between 1 and 10.

Figure 9 compares the performance of disk arrays with MEMS-based storage and several iso-cost architectures. For the sake of clarity, we have shown only one hybrid architecture (LogDisk). The results show that array architectures with MEMS-based storage always exhibit lower latencies than purely disk-based ones, even when the number of disk spindles is increased. The maximum throughput offered by MEMS-based arrays is also substantially higher than that for DiskOnly architectures.



(a) Small IOs (4KB)



(b) Large IOs (64KB)

Figure 6: Maximum throughputs for LogDisk variants with increased numbers of disks. LogDisk+K has K additional disks.

the LogDisk architecture. Adding extra disks for the log improves the maximum throughput when the workload consists mostly of large writes and the demand for disk bandwidth for log-writes is the greatest.

4.3.3 Comparisons using application traces

In order to be able to saturate all array architectures by replaying traces, we varied the intensity by scaling all inter-arrival times in the traces by a constant factor. Hence, the scale factor of one corresponds to replaying the trace at its original speed; the scale factor of two corresponds to replaying the trace twice as fast, and so on.

Figures 7 and 8 present the iso-capacity results, where equal capacity MEMS storage is used for the disks they replace. The MEMSdisk architecture had the highest maximum throughput for all the workloads studied and the lowest latencies for all but one (*tpcd*). Compared with the DiskOnly architecture, MEMSdisk decreased the mean I/O latency by a factor of between 4.0 and 6.5 at the knee of the latency curve for the DiskOnly architecture and increased the maximum throughput by a factor of between 4 and 28.

As expected, the hybrid architectures (MEMSmirror, DualStripe, and LogDisk) have a performance between that of DiskOnly and MEMSdisk. Maximum throughput ranged between 3 and 20 times that of DiskOnly. Among the hybrid architectures, LogDisk had the best performance for three of the four traces: *cello*, *omail* and *tpcc*. While all of the hybrid architectures improve read throughput by accessing the data in the MEMS copy, only the LogDisk architecture offers a significantly higher write throughput for a wide range of workloads

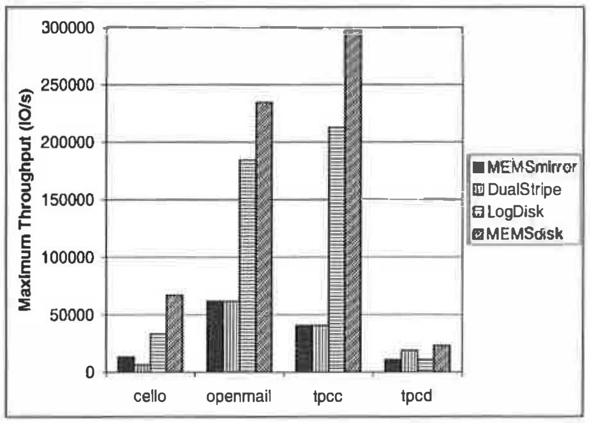


Figure 7: Maximum throughputs for hybrid architectures for the trace workloads.

(Section 4.3.2). For the *tpcd* workload, DualStripe had the highest throughput among the hybrid architectures and an I/O latency even lower than MEMSdisk. The aggressive prefetching behavior of DualStripe is particularly well suited to this workload, which exhibits mainly large sequential reads.

Replacing the NVRAM cache in a conventional disk array with a (much larger) MEMS cache is effective in reducing average response time, but does not improve throughput. For the original trace replay speed, MEMScache was able to reduce the I/O latency between 23% and 82%; this improvement in I/O latency is far lower than in the architectures where MEMS devices stored at least one copy of the data. The MEMS cache is ineffective for cold misses for the read operations, which contributes substantially to the latency. Although the MEMS the parity). Given that most of the architectures we introduced have a higher cost per byte than DiskOnly, it is legitimate to ask what the performance of DiskOnly would be if the extra money spent on MEMS were to be spent on additional disks instead, to get more spindles in the backend. If the data is striped over all disks, there are two potential performance advantages: more disk arms imply more potential parallelism, and partially-empty disks incur shorter seeks. To address this question we studied the *Isocost-X* architectures, *i.e.*, instances of DiskOnly in which the number of disk drives is increased until the cost matches that of a MEMSdisk architecture, assuming that the per-byte cost ratio of MEMS storage to disk is X.

4.3.1 Synthetic workloads

Our first set of experiments were designed to outline the performance of all the architectures studied. We used synthetically-generated workloads, which are easily scaled, to determine the maximum throughput and normal-usage of each architecture. The maximum throughput is found by measuring throughput with an offered load of 1 million IO/s, which is well above the throughput limit of any of the architectures. We then measured the *normal-usage* latency at 50% utilization by using a workload with an IO rate equal to half the maximum throughput.

Figure 5 shows the measured maximum throughput and normal-usage latencies for all the architectures studied. As one would expect, the MEMSdisk architecture is the clear leader in maximum throughput, with 380,000 IO/s, a 20-fold increase over the DiskOnly architecture. Among the hybrid (disk+MEMS) architectures, the LogDisk architecture is the best, with an approximately 177,000 IO/s maximum throughput; the MEMSdisk and the DualStripe architectures had substantially lower performance, with around 51,000 IO/s, as they are not as efficient as the LogDisk architecture for writes. In the DiskOnly-X iso-cost architectures, which increased the number of disk spindles to match the cost of the MEMSdisk architecture, the maximum throughput increased with the cost factor X, but even with a MEMS/disk cost factor of $X = 10$, the throughput was approximately half that for the MEMSdisk and slightly less than that for the LogDisk. The MEMScache architecture that replaced the NVRAM cache with a MEMS cache, improved the performance only slightly as the larger, MEMS-based cache was not effective for this workload.

The normal-usage latency numbers reflected the presence (or absence) of MEMS in the architecture: the DiskOnly architectures show a latency of 2.5–5.4ms, whereas the architectures using MEMS storage show a latency of 0.7–1.1ms.

We conclude that arrays using MEMS storage will offer substantially higher throughputs and lower latencies than those using disks alone, even if the number of disk spindles is increased. The hybrid architectures, which combine disks and MEMS devices, improved the IO latency significantly, but only the LogDisk showed a significant improvement in throughput for the synthetic workloads. In the next section, we examine the LogDisk architecture more closely to better understand its performance characteristics.

4.3.2 Comparing LogDisk variants

Our hybrid disk/MEMS architectures are predicated on the assumption that, with an appropriate layout and access policies, mirroring data on MEMS storage and disks can provide most of the performance of purely MEMS-based arrays at a lower cost. By default, we used MEMS storage and disks of equal capacity; however, it takes several MEMS chips to equal the capacity of a single disk, and these MEMS chips can sustain a higher aggregate IO rate than the disk. For the 36 GB disks we used in our experiments, we used 15 MEMS chips; each disk was able to sustain about 250 IO/s and each MEMS chip were sustaining about 1000 IO/s with 4K random requests. To determine whether increasing the number of disks would improve the performance of hybrid architectures, we compare the maximum throughput obtained from a LogDisk architecture with a varying number of disks. We use synthetic workloads with varying fractions of reads, for small (4KB) IOs and large (64KB) IOs.

Figure 6 shows the throughput obtained. The baselines are MEMSdisk (with only MEMS storage) and LogDisk (with equal capacities of MEMS and disk storage). The LogDisk architecture has two disks and 30 MEMS chips organized into two logical disks of 15 chips each. LogDisk-K represents a LogDisk variant with 2+K disks. For small IOs, the maximum throughput in LogDisk does not improve significantly when the number of disks is increased. Since the LogDisk uses large sequential writes to the disks with a bandwidth of about 125 MB/s per disk, disks do not become a bottleneck and the additional disks provide no performance benefits. For large IOs, the maximum throughput improves, especially for workloads with mostly writes, when the number of disks is increased by 2; beyond that, the improvements due to increasing the number of disks are small. For workloads with at least 50% reads, there is little difference between the various LogDisk variants.

Overall, we conclude that providing a number of disks to match the capacity of the MEMS storage is adequate for made extremely accurate, up to the extreme of running the code from the corresponding component's firmware. To exercise the simulated system, we had Pantheon generate synthetic workloads, and replay traces taken on real systems. In the configurations we used, Pantheon issued each I/O at the same time it was issued in the original trace (for the same replay speed), regardless of whether previous accesses had completed or not.

Our instantiations of Pantheon contain MEMS modules based on the state-of-the-art performance model described by Sivan-Zimet and Madhyastha [21]. We configured those modules to simulate a conservative version of the first generation of MEMS-based device characteristics used by Schlosser *et al.* [20]. Table 1 contains the parameters for the MEMS-based storage device characteristics used in our study. We also updated the disk models in Pantheon to simulate a disk drive based on an aggressive extrapolation of performance characteristics of modern high-performance disks — 3 ms average seek time, 20K rpm rotational speed, and a transfer bandwidth of 125 MB/s to/from the media. We configured the simulator to use four back-end buses to connect the disks to the disk array controller, we used an extrapolated bus bandwidth of 1GB/s for our simulations.

Our cost-performance comparisons used a cost of $6/GB for disks, based approximately on 2001 list prices for enterprise-class disks [17] and $8/MB for NVRAM, based on recent Dallas Semiconductor list prices. Since the cost for MEMS is unknown, we varied the relative per-byte cost of MEMS storage to disks between 1 and 10.

4.2 The workloads

In our evaluation, we used both synthetic workloads and several application traces: a file server containing home directories of a research group (*cello*), an e-mail server for a large company (*omail*), a database server running an on-line decision-support benchmark (*tpcd*), and a transaction processing benchmark (*tpcc*).

The filesystem trace (*cello*) represents one hour of user activity on April 20, 1999 on our main file server at HP Labs. The server stored a total of 63 filesystems containing user home directories, news server pools, customer workload traces, HP-UX OS development infrastructure, etc., for a total of 238 GB user data in a 479 GB physical storage. This is a typical I/O workload for a research group, involving software development, trace analysis, simulation, e-mail, etc. This trace has 370,000 I/O requests, with an average size of 23KB.

The *omail* workload is taken from the trace of accesses done by an OpenMail e-mail server [10] on a 640GB message store; the server was configured with 4487 users, of whom 1391 were active. The *omail* trace has 1.1 million I/O requests, with an average size of 7KB.

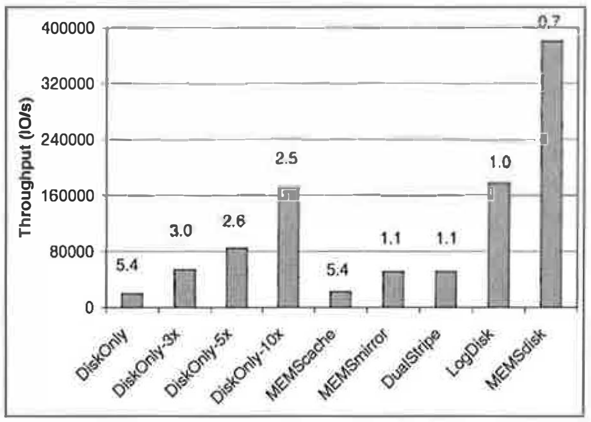


Figure 5: Performance comparison of architectures using synthetic workloads. Numbers above bars show normal-usage latency in *ms* (taken at 50% utilization).

The *tpcd* workload represents decision support systems; it consists of queries 5 and 7 from the TPC-D benchmark at the 300GB scale factor. This benchmark displays long complex database queries with both sequential and random accesses. The *tpcd* trace has 71,000 I/O requests, with an average size of 56KB.

The *tpcc* workload represents on-line transaction processing environments. It is based on a mid-range TPC-C benchmark configuration using one disk array and a two-processor server; overall, the transaction rate was 16.5K tpmC. The *tpcc* trace has 4.2 million I/O requests with an average size of 2KB.

We also used synthetic workloads in our experiments. Request sizes were drawn from an exponential distribution with a mean of 4KB, start addresses were drawn from a uniform distribution over the entire available device address range. The workloads had a varying ratio of read and write requests with 67% reads and the 33% writes as the default ratio, and a request inter-arrival times from an exponential distribution with a variable mean to simulate a variety of workload access intensity.

4.3 Results

Since MEMS-based devices have the potential to affect both the throughput and latency characteristics of disk arrays, we consider both performance metrics. Our baseline is the conventional DiskOnly architecture, *i.e.*, the combination of NVRAM cache and disk back-end found on current disk arrays, with a 2 GB of raw NVRAM and a 2 TB raw physical disks. For the MEMScache, we used 100 GB of logical MEMS storage (120GB raw including

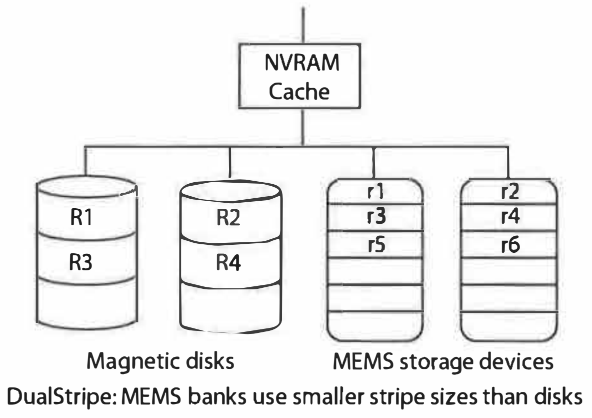


Figure 4: DualStripe hybrid array. Data is mirrored, one copy on magnetic disks with large stripes, another on MEMS storage devices with a small stripe size. Total capacity is equal on MEMS and disk.

RAID 0 layout and store one copy of the data stored on the array; the disks similarly form a RAID 0 group and store another copy. The stripe unit size for the MEMS RAID 0 group is small, to distribute accesses evenly and avoid hot-spots. The stripe unit size for the disk RAID 0 group is large, to reduce the positioning time cost for large sequential reads or writes.

Data written to the array is stored in the NVRAM write cache, to be flushed to the copies on MEMS and on disk when the devices are idle, or when forced because the cache's high-water mark is reached. Read data are obtained from the cache if present there. If not, they are read preferentially from the MEMS copy if the queue is short and from the disk copy if the queue length exceeds a threshold. However, if the read is detected to be part of a sequential run, the data are read from disk and a large subsequent block is prefetched to serve future requests in this sequential run. We expect this architecture to perform well for workloads where a substantial fraction of the reads are sequential.

Sequentiality detection works as follows: the array controller keeps a record of the addresses of the last $sequentialityWindow$ read requests. When a new read request arrives, this record is checked to see if the $sequentialityThreshold$ data blocks sequentially previous to this have been recently accessed. If so, the request is treated as part of a sequential run. In our implementation, we used $sequentialityWindow = 300$ and $sequentialityThreshold = 32KB$.

3.5 MEMScache: MEMS as array cache

The array architectures described so far explore the use of MEMS as a part of a redundancy scheme: for example, to store one of a pair of data replicas. In this section, we look at the other alternative: use of MEMS as a replacement for the NVRAM cache. Any redundant organization can be used for the disk back-end; in our implementations we have assumed a RAID-1/0 layout.

The operation of the MEMS primary cache is similar to that of the usual NVRAM cache. Reads are served from the cache if the data is already present in the cache; otherwise, the data are fetched from disk, and kept in the cache until flushed. Writes are saved in the cache, to be flushed in the background; usually, there are two copies of a data in the cache for redundancy. The dirty data is flushed to the back-end disk drives, when the amount of dirty data in the cache reaches a high-water mark; flushing continues in the background until the remaining dirty data is less than a low-water mark. The flushing process considers the location of dirty blocks in the disk storage so that: (a) the dirty blocks with continuous addresses are aggregated to be flushed in larger chunks and (b) the dirty blocks are written in ascending order of the addresses so that the access pattern for the flushes at the back-end is as close to sequential as possible.

The MEMS cache is a RAID-5 array of MEMS storage devices, organized as a log of cache lines. When data is written into the cache (whether due to a read from disk or an external write), one or more cache lines are appended to the log. If those addresses existed in the cache already, the corresponding locations are marked empty and later reclaimed by a log-cleaning process.

4 Experimental evaluation

We compared the performance and the cost/performance of the proposed architectures against a DiskOnly architecture using synthetic workloads and application IO traces. Performance comparisons are made by ignoring cost and looking at the proposed architectures configured to have equal capacity: we call these the *iso-capacity* comparisons. Cost performancc is comparcd in two sets of experiments: by comparing the performance of MEMSdisk and DiskOnly architectures configured to cost same (*iso-cost* comparisons) and by comparing the cost/performance ratio of selected configurations of all architectures. Since MEMS-based storage chips are not currently available, we made these comparisons using a detailed simulator. We present the experiments and the results below.

4.1 Evaluation environment

We used a detailed event-driven storage system simulator called Pantheon [26]. Pantheon contains independent modules for separate components of the storage system, such as disks, controllers, non-volatile caches, array controllers, and buses. Each module's simulation can be

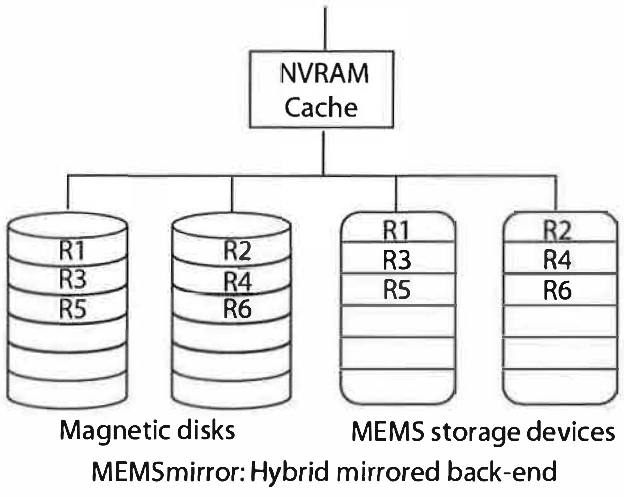


Figure 2: MEMSmirror: each disk is mirrored by a MEMS bank of equal capacity

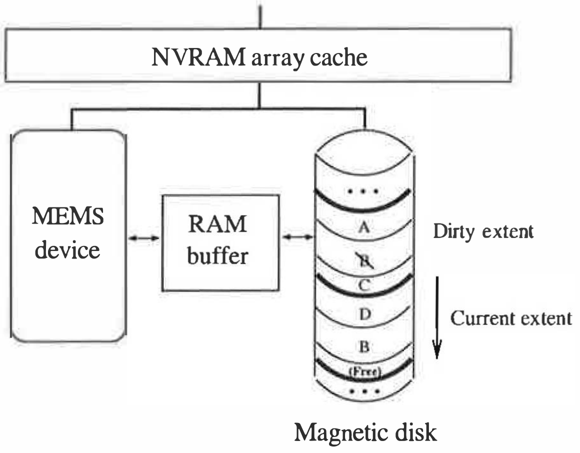


Figure 3: LogDisk architectural diagram: the RAM buffer allows asynchronous transfers between MEMS and disks. The current extent on disk is used in append-only mode, from top to bottom in the figure. Newer writes may render old data (*e.g.*, block B) invalid.

positioning times of the disk copy, data is written to disks as a log for near-to-zero positioning latency. Our disks are standard—they follow the standard update-in-place allocation policies, but we use them in a mostly append-only fashion.

Consider an array of n MEMS storage devices M_1, M_2, …, M_n, plus "mirror" disks D_1, D_2, …, D_n, of greater capacity than the corresponding MEMS devices. The MEMS storage devices are organized as a RAID 0 (stripe only, no redundancy) array [4]. Other components include NVRAM for saving metadata and for temporary storage of writes, and a RAM buffer for copying data between the MEMS storage devices and the disks. All data reads are serviced from the MEMS array. Writes are inserted into the NVRAM write cache and later flushed to both the MEMS array and the magnetic disk copies (when both copies are written, the data is cleared from the write cache.) Figure 3 depicts a simplified version of the LogDisk architecture, containing a single pair of devices.

Updates to the data stored in M_i are mirrored in D_i in a log-structured fashion. The disk writing algorithm is designed to minimize the fraction of time the disk head spends idle or seeking. The space in each disk is divided into fixed-size extents, one of which is marked as *current*. To write data on the disk copy, the array creates a fresh (*active*) copy of each overwritten block at the end of the current extent, and updates metadata stored in NVRAM to reflect the new status. Disks are therefore mostly used in sequential mode to append new data, sustaining their peak transfer rates with minimal positioning overhead. An I/O operation which overwrites a data block may supersede portions of a previously-written extent on disk, thus making the older extent sparsely populated with active data. Such extents are transformed into usable empty space by cleaning. At most one extent is being cleaned at any point in time. During idle periods the data corresponding to active blocks is read from the MEMS devices into the RAM buffer; whenever the disk is idle (*i.e.*, there is no write data in the write cache to be written to disk), it appends this data to the current extent, marking the corresponding data block in the extent being cleaned as invalid. An extent is *dirty* as long as it has active blocks; it becomes clean when no active blocks remain in it. The operation of the log-structured disk is quite similar to that of the log-structured file system LFS [18].

3.4 DualStripe: Hybrid replication for multiple access types

When redundancy is provided in a storage array by replicating the data, the replicas can be stored in different ways to optimize performance. In the LogDisk architecture described above, the disk copy is organized as a log to minimize the cost of writes; reads are directed primarily to the MEMS copy. However, disks can perform sequential reads very efficiently; we now describe an architecture in which the disk copy can service sequential accesses. The DualStripe architecture dynamically detects the sequentiality characteristics of the workload, and services accesses from the devices that are best suited for them according to the recent access history.

Consider an array with n MEMS storage devices, m magnetic disks, and a mirrored NVRAM write cache (Figure 4). The MEMS devices are organized in a

bit width (nm)	50
sled acceleration (g)	70
access speed (kbit/s)	400
settling time on x (ms)	0.431
total tips	6400
simultaneously active tips	640
max. throughput (MB/s)	25.6
number of sleds	1
per-sled capacity (GB)	2.56

Table 1: MEMS-chip parameters.

but their variances are also much lower.

Table 1 summarizes the parameters of the MEMS-based storage chips we used. These parameters correspond to conservative predictions [20] for the characteristics of the first generation of MEMS-based storage chips. Our simulated chips do not allow bidirectional reads, *i.e.*, accesses along the y axis must always be done while moving the sled in the same direction.

3 MEMS-based array architectures

Current high-end disk arrays store data in two main locations. They typically contain a fully-associative NVRAM cache in the order of tens of gigabytes. User data is ultimately stored in the *back-end* disk drives, for a total capacity of many terabytes. For fault tolerance, arrays keep redundant data at both levels in the memory hierarchy: as mirror copies or erasure-correcting codes on disks (RAID) [1, 16], and as dirty blocks mirrored in separate NVRAM cache banks in independent power domains. Disk arrays organize data storage into *Logical Units* (LUs), exporting a linear address space of blocks to client hosts.

The NVRAM cache in the disk arrays serves several purposes. First, it acts as a speed-matching buffer between the disks and storage area networks. Second, it allows the array to report the completion of write accesses as soon as the dirty data is in the (fault-tolerant) cache, without waiting for the disk write to complete. This optimization, commonly known as write-behind, decreases I/O service times and allows writes to be performed more efficiently in the background. Third, the NVRAM cache exploits the temporal locality in the workloads: multiple overwrites on the same data in the NVRAM cache are folded into a single write to the back-end during destaging; similarly, multiple read accesses to the same data can be directly served from the cache. Finally, read-ahead optimizations can exploit spatial locality in the workloads.

Our primary goal is to propose and evaluate alternative ways in which MEMS-based storage could improve both the performance and the cost/performance ratio of current disk arrays. We study architectures that use MEMS as either a total replacement for all back-end disks, or as a replacement for only some of them (hybrid architectures), or as a total replacement of current NVRAM cache. The hybrid architectures we have studied include several different data layouts and corresponding IO access policies, in order to determine if the different characteristics of disks and MEMS storage can be exploited for better performance. Despite the obvious fact that many other ways exist to incorporate MEMS into storage architectures, this methodology includes multiple points across the cost/performance spectrum for a reasonable degree of coverage of the potential alternatives.

3.1 MEMSdisk: Array disk replacement

The MEMSdisk architecture replaces each disk drive in the disk array by a bank of MEMS-based storage devices of the same capacity. Since the access latencies are much smaller for MEMS-based devices, the MEMSdisk architecture provides an upper bound on performance for all arrays that utilize MEMS-based storage for a fixed cache size. However, this comes at a potentially high cost per byte—up to an order of magnitude more expensive than disk drives of the same capacity.

3.2 MEMSmirror: Hybrid mirrored back-end

A RAID1/0 Logical Unit(LU) in a conventional disk array comprises a number of disk pairs where both disks in each pair contain exactly the same data. Writes complete as soon as they are written to the redundant NVRAM cache. The data is later flushed to the disks in the background, when the disks are otherwise idle, or when the cache starts filling up. Reads of data not found in the cache, however, require disk accesses, which have substantial latency.

MEMSmirror, depicted in Figure 2, alleviates this problem by having hybrid mirrored pairs: one disk drive and one bank of MEMS storage of the same capacity. Reads of data not in the cache are directed to the MEMS copy, which has much lower latency and higher throughput than the disk copy. Since the disk copy only handles writes, it can sustain a fairly high throughput, and the disk latencies are not an issue because of the NVRAM cache.

3.3 Logdisk: Hybrid replication with log-structured disk storage

As an attempt to get as close as possible to the performance of a purely MEMS-based array without the consequent cost, we propose an alternative where the data in MEMS storage devices is mirrored for redundancy on magnetic disks. In order to diminish the impact of slow dundancy schemes, caching methods, partial and complete replacement of disk and NVRAM with MEMS storage, variation in the proportions of MEMS storage, disk and NVRAM, as well as combinations of these methods. This study is intended to narrow the focus of future explorations by finding a few disk array architectures where the use of MEMS is most beneficial. We do this by devising a number of novel architectures to examine the potential placements of MEMS-based storage in a disk array. We concentrate on the performance part of users' requirements [28], as no predictions are yet available of the reliability and availability characteristics of MEMS chips—not even the cost of mass-produced chips is known precisely. Given that MEMS-based storage chips will not be commercially available before 2004, we used a detailed simulator replaying I/O traces from real applications for our performance study. By providing insight into the various architectural tradeoffs, our resulting cost/performance analysis can be seen as a first-cut indication on where to best spend money when designing disk arrays using MEMS-based storage devices. We found that replacing disks with MEMS storage in disk arrays improves both the performance and the performance/cost significantly, even if the MEMS storage costs ten times as much per byte as disks do. We also found that some hybrid MEMS/disk architectures offer an intermediate performance and cost between conventional disk arrays and MEMS-based arrays, with a performance/cost similar to MEMS-based arrays.

The remainder of this paper is organized as follows. Section 2 gives an overview of MEMS-based storage devices. We describe the disk array architectures under consideration in Section 3, and evaluate their performance and cost/performance characteristics in Section 4. Section 5 surveys related work; Section 6 contains a final discussion.

2 MEMS-based storage basics

MEMS-based storage chips consist of arrays of scanning probe tips that access a rectangular storage media sled. MEMS storage chips are built using standard photolithographic CMOS processes, and are expected to be massively produced around 2004. While the final design parameters for MEMS-based storage chips are still an active area of study, we concentrate on high-level device characteristics, as they relate to the present work. Carley, Ganger and Nagle [3] and Griffin *et al.* [8] provide detailed descriptions of MEMS-based storage.

As shown in Figure 1, MEMS-based storage chips contain one (or more than one, depending on design decisions [6]) rectangular array of several thousand probe tips. Data is stored on a rectangular media sled that moves in two dimensions with respect to the array of tips. We study the prevalent variety in which data storage is magnetic, as in disk drives; other recording materials such as phase-change media as in re-writable CDs are also possible. Each probe tip accesses a rectangular region on the media sled, and no two such regions overlap. For a given access, the media sled simultaneously seeks in the x and y directions until the corresponding tip is positioned on top of the start bit; then, the sled keeps moving in the y direction, while reading or writing consecutive bits along its trajectory. Since multiple probe tips can be active at any given time, most proposed data layouts rely on bit-interleaving, with multiple tips performing parallel reads or writes.

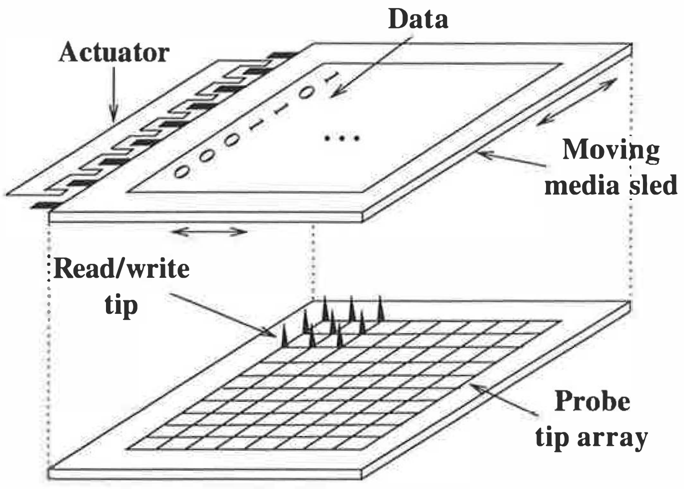


Figure 1: High-level schematic of a MEMS-based storage chip. All four sides of the media sled have actuators, and every crossing point in the tip array has a read/write tip. The sled is supported on top of the array by an ensemble of cantilevered springs, that move in two dimensions to seek to the coordinates where the data are.

This design has several important consequences. First, stored data is persistent and does not depend on continuous availability of a power source, as in battery-backed DRAM caches. Second, positioning delays depend on the relative positions of the sled and of the destination coordinates. Third, positioning delays are much smaller than in disk drives as there are no rotational delays, ranges of motion are in the order of a few millimeters, and components have small masses. Schlosser *et al.* found, using simulation [20], that typical access times for MEMS are in the order of 1–2 ms. The advantage over disk drives is still more pronounced for random workloads, where the disk spends most of the time positioning the head over the right bits instead of actually transferring data to/from the platters. Thus, MEMS positioning times are not only smaller on average than those of disks,

Using MEMS-based storage in disk arrays

Mustafa Uysal   Arif Merchant   Guillermo A. Alvarez
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304, USA

Abstract

Current disk arrays, the basic building blocks of high-performance storage systems, are built around two memory technologies: magnetic disk drives, and non-volatile DRAM caches. Disk latencies are higher by six orders of magnitude than non-volatile DRAM access times, but cache costs over 1000 times more per byte. A new storage technology based on microelectromechanical systems (*MEMS*) will soon offer a new set of performance and cost characteristics that bridge the gap between disk drives and the caches. We evaluate potential gains in performance and cost by incorporating MEMS-based storage in disk arrays. Our evaluation is based on exploring potential placements of MEMS-based storage in a disk array. We used detailed disk array simulators to replay I/O traces of real applications for the evaluation. We show that replacing disks with MEMS-based storage can improve the array performance dramatically, with a cost performance ratio several times better than conventional arrays even if MEMS storage costs ten times as much as disk. We also demonstrate that hybrid MEMS/disk arrays, which cost less than purely MEMS-based arrays, can provide substantial improvements in performance and cost/performance over conventional arrays.

1 Introduction

Disk arrays [16] are the main building blocks used to satisfy the performance and dependability requirements of current high-end storage systems. A disk array consists of a large number of disk drives, partially used to store redundant data that will allow transparent recovery from disk failures; controllers that interface with client hosts and maintain redundant data; and large battery-backed, non-volatile RAM (NVRAM) caches that allow optimizations such as prefetching, write-behind, and background destaging to mitigate the effects of high disk latencies. Most modern disk array architectures are based on the two-level NVRAM/disk hierarchy.

The access latency gap between disk and NVRAM is currently almost six orders of magnitude (10 ms vs 50 ns), and is widening by about 50% per year. NVRAM costs about three orders of magnitude more per byte than disk drives. While specific applications may enjoy high hit rates from array caches, Wong and Wilkes [25] show that in most cases, NVRAM caches in high-end arrays can only hold 5% of the working set of applications, leading to low hit ratios. NVRAM is much less reliable than disk drives: typical mean time to failure for battery-backed NVRAM is only about 15K hours, compared to over a million hours for disk drives [19]. As a result, almost all disk arrays keep at least two copies of all dirty data in separate NVRAM buffers, further increasing cost. Finally, battery packs are cumbersome, as they must be capable of supplying enough power for the whole array; they can reach hundreds of pounds in weight and many cubic feet in size.

A disruptive new storage technology based on microelectromechanical systems (*MEMS*) will soon offer a new set of performance, cost and reliability characteristics that bridge the gap between NVRAM and disk drives. MEMS-based storage consists of chips containing thousands of small, mechanical probe tips that access data located on flat rectangles of storage media. The media is moved in two dimensions over fixed probe-tip heads, until the desired bits coincide with the heads. Positioning delays for MEMS-based storage are much smaller and more deterministic [8] than those of conventional disk drives. First, there is no rotational delay component in the positioning times. Second, MEMS-based storage is expected to achieve much higher densities (260–720 Gbit/in^2) [3], so seek distances are much shorter than in disk drives. Finally, since moving parts have much smaller masses than those in disks, they are much easier to accelerate. As a result, MEMS-based storage has the potential to bridge the cost and performance gaps between disk drives and NVRAM.

We explore the cost/performance implications of incorporating MEMS-based storage into disk array architectures. The total space of possible disk array architectures is too large to be explored systematically: the possibilities include the use of different data layouts, re-

Guillermo A. Alvarez is now with IBM's Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA. Email: {uysal,arif} @hpl.hp.com, alvarezg@almaden.ibm.com.

[22] W· C. Hsieh, D. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[23] G. F. Hughes. Wise Drives. *IEEE Spectrum*, August 2002.

[24] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[25] R. King. Dirty filesystem bug in 2.4.9-21 ext3. https://listman.redhat.com/pipermail/ext3-users/2002-April/003343.html, March 2002.

[26] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[27] M. K. McKusick, W· N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[28] A. Morton. Data corrupting bug in 2.4.20 ext3. http://www.uwsg.iu.edu/hypermail/linux/kernel/0212.0/0010.html, Dec. 2002.

[29] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

[30] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *SIGCOMM 2001*, San Deigo, CA, August 2001.

[31] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, CA, June 2002.

[32] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *VLDB*, New York, NY, August 1998.

[33] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[34] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon, 1999.

[35] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[36] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.

[37] K. Swartz. The Brave Little Toaster Meets Usenet. In *LISA '96*, pages 161–170, Chicago, Illinois, October 1996.

[38] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[39] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[40] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, February 1999.

[41] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[42] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, October 2000.

[43] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, CA, October 1998.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 307–322, June 2000.

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. *Transactions on Computer Systems (TOCS)*, 20(1), February 2002.

[4] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[5] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, and SIGPLAN.

[6] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *The Tenth USENIX Security Symposium*, Washington, D.C., August 2001.

[7] J. Brown and S. Yamaguchi. Oracle's Hardware Assisted Resilient Data (H.A.R.D.). *Oracle Technical Bulletin (Note 158367.1)*, 2002.

[8] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, CA, June 2002.

[9] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[10] C. S. Collberg. Reverse Interpretation + Mutation Analysis = Automatic Retargeting. In *Conference on Programming Language Design and Implementation (PLDI '97)*, Las Vegas, Nevada, June 1997.

[11] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 15–28, Asheville, NC, December 1993.

[12] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, CA, June 2002.

[13] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[14] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[15] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 237–252, San Francisco, CA, January 1992.

[16] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[17] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[18] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, 1997.

[19] J. Gray. Storage Bricks Have Arrived. Invited Talk at the First USENIX Conference on File And Storage Technologies (FAST '02), 2002.

[20] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *The Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[21] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

EOF Fingerprinting			
Probe process	1061		
In SDS	2542		
SDS Infrastructure		**Case Studies**	
Initialization	395	Traxtents	1320
Hash table and cache	2122	File-aware cache	203
Direct classification	195	Journal cache	305
Indirect classification	75	Meta-data cache	235
Association	15	Secure delete	80
Operation inferencing	1105	Journaling	2440

Table 9: **Code Complexity.** *The number of lines of code required to implement various aspects of an SDS are presented. For the "In SDS" component of the EOF tool, there are 2542 lines of code; roughly 1800 of those lines are shared among all file system types; the rest is file-system specific.*

ing file system, ext3. It is only during file creation that the SDS pays a significant cost relative to ext3; the overhead of block differencing and hash table operations have a noticeable impact. Since the purpose of this case study is to demonstrate that an SDS can implement complex functionality, this small overhead is certainly acceptable.

7.2 Complexity Analysis

We briefly explore the complexity of implementing software for an SDS. Table 9 shows the number of lines of code for each of the components in our system and the case studies. From the table, one can see that most of the complexity is found in the EOF tool, the basic cache and hash tables, and the operation inferencing code. Most of the case studies are trivial to implement on top of this base infrastructure; however, the Traxtent SDS and the Journaling SDS require a few thousand lines of code. Thus, we conclude that including this type of functionality within an SDS is quite pragmatic.

8 Conclusions

"Beware of false knowledge; it is more dangerous than ignorance." *George Bernard Shaw*

In a recent article on "Wise Drives", Dr. Gordon Hughes, Associate Director of the Center for Magnetic Recording Research, writes in favor of smarter drives, stressing their great potential for improving storage system performance and functionality [23]. However, he believes a new interface between file systems and storage is required: "For widespread uses, its [a drive's] input/output and command requirements need to appear in the interface specification. In short, there must be an industry consensus that the task is of general interest and offers market opportunities for multiple computer and drive companies." Hughes' comments illustrate the difficulty of new interfaces – they require wide-scale industry agreement, which eventually limits creativity to only those inventions that fit into an existing interface framework.

With information about how the file system uses the disk and low-level knowledge of drive internals, an SDS sits in an ideal location to implement powerful pieces of functionality that neither a disk nor a file system can implement on its own, enabling new innovations behind existing interfaces. Further, storage system manufacturers can now embed optimizations that previously were relegated to the domain of file systems, enabling vendors to compete along axes other than cost and performance.

In this paper, we have demonstrated that underneath of a particular class of FFS-like file systems, file-system information can be automatically gathered and then exploited to implement functionality in drives that heretofore had to be implemented in the file system or could not be implemented at all. We have shown that the costs associated with reverse-engineering file system structure and behavior are reasonable.

Many challenges remain, including understanding the generality and robustness of semantic inference across a broader range of file systems. Can more sophisticated file systems across a wider range of platforms be probed to reveal their inner workings? Can approximate information be further exploited to implement interesting new functionality? Can more techniques and tools be developed to assure the correct operation of semantic technology? We believe the answer to these questions is yes, but only through further research and experimentation will the final answer be elicited.

Acknowledgments

We would like to thank the members of the WiND research group for their feedback on the ideas presented within this paper. We also would like to thank Keith Smith for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions, many of which have greatly improved the content of this paper. Finally, we thank the Computer Systems Lab for all of their tireless assistance in providing a terrific environment for computer science research. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, an IBM Faculty Award, and the Wisconsin Alumni Research Foundation. Timothy E. Denehy is sponsored by an NDSEG Fellowship from the Department of Defense.

	Delete	PostMark
ext2	24.0	103.0
+Secure-deleting SDS_2	46.9	128.0
+Secure-deleting SDS_4	56.9	142.0
+Secure-deleting SDS_6	63.6	192.0

Table 7: **Secure Deletion.** *The table shows the time in seconds to complete a Delete microbenchmark and the PostMark benchmark on the Secure-deleting SDS. The Delete benchmark deletes 1000 32-KB files, whereas the PostMark benchmark performs 1000 transactions. Each row with the Secure-deleting SDS shows performance with a different number of over-writes (2, 4, or 6). This experiment took place on "slow" system running Linux ext2 mounted synchronously upon the IBM 9LZX disk.*

	Create	Read	Delete
ext2 (2.2/sync)	63.9	0.32	20.8
ext2 (2.2/async)	0.28	0.32	0.03
ext3 (2.4)	0.47	0.13	0.26
ext2 (2.2/sync)+Journaling SDS	0.95	0.33	0.24

Table 8: **Journaling.** *The table shows the time to complete each phase of the LFS microbenchmark in seconds with 1000 32-KB files. Four different configurations are compared: ext2 on Linux 2.2 mounted synchronously, the same mounted asynchronously, the journaling ext3 under Linux 2.4, and the Journaling SDS under a synchronously mounted ext2 on Linux 2.2. This experiment took place on the "slow" system and the IBM 9LZX disk.*

overwrite until the disk is idle, instead of performing it immediately; freeblock scheduling may further minimize the performance impact [26].

Journaling: The final case study is the most complex – the SDS implements journaling underneath of an unsuspecting file system. We view the Journaling SDS as an extreme case which helps us to understand the amount of information we can obtain at the disk level. Unlike most of the other case studies, the Journaling SDS requires a great deal of precise information about the file system.

Due to space limitations, we only present a brief summary of the implementation. The fundamental difficulty in implementing journaling in an SDS arises from the fact that at the disk, transaction boundaries are blurred. For instance, when a file system does a file create, the file system knows that the inode block, the parent directory block, and the inode bitmap block are updated as part of the single logical create operation, and hence these block writes can be grouped into a single transaction in a straight-forward fashion. However, the SDS sees only a stream of meta-data writes, potentially containing interleaved logical file system operations. The challenge lies in identifying dependencies among those blocks and handling updates as atomic transactions.

As a result, the Journaling SDS maintains transactions at a coarser granularity than what a journaling file system might. The basic approach is to buffer meta-data writes in memory and write them to disk only when the in-memory state of the meta-data blocks constitute a consistent meta-data state. This is logically equivalent to performing incremental in-memory fsck's on the current set of dirty meta-data blocks and writing them to disk when the check succeeds. When the current set of dirty meta-data blocks form a consistent state, they are treated as a single atomic transaction, thereby ensuring that the on-disk meta-data contents either remain at the previous (consistent) state or are fully updated with the next consistent state. One benefit of these more coarse-grained transactions is that by batching commits, performance may be improved over more traditional journaling systems.

To guarantee bounded loss of data after a crash, the Journaling SDS limits the time that can elapse between two successive journal transaction commits. A journaling daemon wakes up periodically after a configurable interval and takes a copy-on-write snapshot of the dirty blocks in the cache and the dependency information. After this point, subsequent meta-data operations update a new copy of the cache, and therefore cannot introduce additional dependencies in the current epoch.

Similar to the Secure-deleting SDS, the current Journaling SDS implementation assumes the file system has been mounted synchronously. To be robust, the SDS requires a way to verify that this assumption holds and to turn off journaling otherwise. Since the meta-data state written to disk by the Journaling SDS is consistent regardless of a synchronous or asynchronous mount, the only problem imposed by an asynchronous mount is that the SDS might miss some operations that were reversed (*e.g.*, a file create followed by a delete); this would lead to dependencies that are never resolved and indefinite delays in the journal transaction commit process. To avoid this problem, the Journaling SDS looks for a suspicious sequence of changes in meta-data blocks when only a single change is expected (*e.g.*, multiple inode bitmap bits change as part of a single write) and turns off journaling in such cases. As a fall-back, the Journaling SDS monitors elapsed time since the last commit; if dependencies prolong the commit by more than a certain time threshold, it suspects an asynchronous mount and aborts.

We evaluate both the correctness and performance of the Journaling SDS. To check correctness, we crashed the file system numerous times, and ran `fsck` to verify that no inconsistencies were reported. The performance of the Journaling SDS is summarized in Table 8. Although this SDS requires the file system to be mounted synchronously, its performance is similar to the asynchronous versions since the semantically-smart disk system delays writing meta-data to disk. In the read test the SDS has similar performance to the base file system (ext2 2.2), and in the delete test, it has similar performance to the journal-

Group ID		Revoked capability ID's
Index	Counter	(bitmap)
0	10	100010001000...
1	5	001111011010...
2	14	111011111000...
⋮	⋮	⋮
63	3	000011010111...

Figure 3: **Keeping track of revocations.** The table used by the disk controller to keep track of revoked capabilities.

in a group, the group ID is removed from the list of valid groups and it is replaced with a new group ID.[3] Then, all capability ID's of the new group are marked valid.

In our particular implementation, we divide a group ID into two parts: a 6-bit index and a 64-bit counter. The index part is used to index a 64-entry table, each entry of which contains a counter and 8,128 bits of revocation data (see Figure 3). The table requires $64 \times (8128 + 64)$ bits or 64 KB of RAM and supports up to $64 \times 8128 = 520,192$ simultaneous capabilities. A capability is checked by looking up the entry corresponding to the index part of its group ID, and verifying that the counter matches the one in the capability's group ID. If so, the bit corresponding to the capability ID is tested. Revocation of a capability is done similarly. Group invalidation is done by clearing the group's bitmap and incrementing its counter, effectively replacing its group ID with a fresh new one. All these operations are very quick (small constant time) and space efficiency is excellent: each capability takes on average less than 1.01 bits.

Note that capability groups alone do not reduce the total work on the metadata server over time, but the work gets spread over a longer period, avoiding the burstiness problem. We have done simulations with real trace data that confirm our predictions. Our simulations show that the peak load on the metadata server is reduced significantly with this technique (see Section 4.1 for more details).

2.3 Network partitions

When a network partition separates the metadata server from a disk, the server is unable to revoke capabilities for that disk, resulting in the access permissions of files on that disk effectively being frozen; in some systems, this could be considered a security breach. To avoid this problem, we can require the metadata server to periodically refresh the table of groups and capabilities of each disk. If a disk does not receive a refresh message within a certain period of time, it disallows all accesses until it receives the expected server refresh.

Of course, such a scheme can be disabled if the system administrator believes that the overhead of the refresh messages is too high for the protection it provides.

[3]Note that the group ID's *cannot* be recycled, which means that in theory the system will eventually run out of space. But by using relatively few bits for the group ID's—say 64 bits—it will take longer than the life of the system for that to happen.

2.4 Preventing replay attacks

While it is infeasible for an adversary to forge new requests, it is trivial to replay requests that have already been sent to a disk. Hence, a NAD file system that operates using an unsecured network must have robust defenses against replay attacks. (Note that replay attack prevention is harder than duplicate detection [2, 12], which assumes all parties are honest.) Fortunately, it is possible to achieve this at low cost in memory and computation, without requiring per-client information. The method, which we believe is novel in the context of replay attacks, employs a data structure called a Bloom filter [4] to remember recent requests.

Bloom filters are a highly efficient way of performing approximate set-membership queries; given a membership query, they answer either "probably an element" or "definitely not an element". A Bloom filter consists of an array of K bits, denoted $b_1, b_2, \ldots, b_K$, together with $n \geq 1$ hash functions, $f_1, \ldots, f_n$. The hash functions are chosen randomly from a family of independent hash functions at filter construction time; each maps requests to integers in $\{1, 2, \ldots, K\}$. The filter is defined to be *empty* when all bits are 0. A request r is added to the filter by setting the bits with indices $f_1(r), f_2(r), \ldots, f_n(r)$—*i.e.*, we set $b_{f_i(r)} = 1$, for all i. To answer the question "is request r in the filter?", we reply "probably" if $b_{f_i(r)} = 1$, for all i, and "definitely not" otherwise.

A disk can detect replays by keeping a list of seen requests in a Bloom filter. When a new request arrives, the disk checks to see if it is already in the filter. If the filter reply is "definitely not", the disk can safely proceed to process the request after adding it to the filter as it cannot be a replay. Otherwise, it is likely that the request has already been issued in the past, so the disk sends a replay rejection message. The client continues to retransmit a request until it receives either an acceptance or rejection message for that request. If it gets a reply rejection message, it changes the request's nonce (so that it hashes differently) and continues retransmitting. For the nonce, we use a small sequence number, which serves no other purpose. Note that in a system with message losses, a client may sometimes end up executing its own request multiple times consecutively, but this is not a problem when requests are idempotent.

Of course, after enough requests have been added, the filter will begin to have a non-negligible false-positive rate. We consider a filter in need of replacement when more than a fixed proportion of its bits are set. We implement filter replacement by maintaining several filters at the disk together with a monotonically-increasing *epoch number*, which is periodically checkpointed to disk. Each filter is associated with a recent epoch. When the filter corresponding to the current epoch needs replacement, the disk increments its epoch number, deletes its oldest filter, and starts a new filter to handle the new epoch. On reboot, the epoch number is incremented by the number of filters; this prevents replaying messages sent while the disk was down.

A client sends what it believes to be a disk's latest epoch number—each disk message includes the current number—with every message to the disk. If the epoch number in a client request is too old (*i.e.*, more out of date than the number of filters being maintained), the request is rejected. Otherwise, it is checked against the appropriate Bloom filter. In this way, the switch to a new filter can be made transparent to active clients of the disk. (Clients idle sufficiently long will have their first request rejected due to its out-of-date epoch number.)

It is worth pointing out the following optimization: instead of applying the hash functions to the whole request r, which can be quite large (*e.g.*, it includes the data in a write operation), it suffices to apply them to just the request MAC $m = h(op, s)$ (described in Section 2.1), which is only a few bytes long. Note that the request epoch number and nonce are included in the operation op, which is guarded by the MAC, preventing an attacker from altering them.

Another optimization involves not storing read requests in the Bloom filter, allowing for even smaller filters. Note that read requests need only be checked if encryption is turned off. And even in that case, it may not be necessary to check *recent* read requests, because the attacker could have snooped on the reply of the original read. Thus, only very old read requests need to be filtered out, and this can be accomplished by simply verifying that the request's epoch number is valid; there is no need to use the Bloom filter at all. (Epoch numbers should be periodically advanced with this optimization.) Our performance numbers do not include this optimization.

Our method to prevent replay attacks with Bloom filters is simple, robust, and frugal. In contrast, existing methods such as [3], which keep per-client state, have two drawbacks: (1) they can support only a limited number of clients when constrained to use similarly small amounts of memory, and (2) they require the extra complexity of authenticating clients to the disk to guard against a rogue client claiming too large a share of the client-state table.

Our approach is also different from NASD, which relies on a real-time disk clock and expiration times instead of an epoch number that can be bumped at any time. Unfortunately the NASD scheme limits the maximum rate of requests that NASD can handle to the maximum size of its recent-request list (stored using a less space-efficient array) divided by its expiration time [9]. This could be a problem if many requests hit the disk cache.

2.5 Consistency attacks

If leases (*i.e.*, locks with timeouts) are used to cache filesystem data at the clients, an attacker that wishes to create consistency problems can proceed as follows: A client gets a lock for a file and issues a write request. The attacker then launches a denial-of-service attack to simultaneously capture and obliterate the write request (and subsequent retries) so that it never reaches the disk. After the lock has expired, the attacker sends the captured write request to the disk, which executes the write without the lock held, potentially causing consistency problems.

To guard against this type of attack, the system could invalidate requests that are outstanding when the lock expires. To do that, the metadata server could revoke all capabilities issued to the client that holds the expired lock; the metadata server then waits until the disk has acknowledged the revocations before it breaks the lock.

2.6 Data structures and disk functionality

Our addition of security to a NAD file system requires several data structures, which are listed in Figure 4. At the metadata server, we maintain a hash table of all valid capabilities for use in performing revocations: whenever the access to a file changes, we need to find all capabilities associated with that file and revoke them. The server also maintains copies of each disk's valid group list plus the number of valid and revoked capabilities in each group so that it can quickly choose which group to invalidate next.

At the metadata server:

- hash table of all currently valid capabilities, indexed by inode number
- for each disk, a list of valid groups, with the number of valid and revoked capabilities in each

At a client:

- a cache of capabilities issued to this client that are not known to have been revoked

At a disk:

- one counter and bitmap per valid group (64 KB)
- Bloom filters of recent requests (64 KB)

Figure 4: **Additional data structures for security.**

new functionality	equivalent lines of C
cryptography	340
capability groups and revocations	60
miscellaneous (refresh timer, RPC handlers, logging)	610

Figure 5: **Additional disk functionality.** The left column describes the purpose of the additional functionality that would be required on a secure disk; the right column gives the number of lines of C devoted to that functionality in our software implementation.

Clients cache issued capabilities to cut down on metadata-server traffic. No invalidation protocol is needed because if a client uses a cached capability that is no longer valid, the disk will reject it, leading the client to request a new one from the metadata server. The data structures at a disk have already been discussed in the sections on capability management (Section 2.2) and replay attacks (Section 2.4).

Figure 5 lists the modest extra functionality required to add our block-based security to a NAD. Combined with the fact that the additional data structures could use as little as 128 KB of RAM (see Figure 4), this suggests that our approach requires minimal changes to disks.

3 Implementation

We have implemented a prototype NAD file system, called *Snapdragon*, that uses our security approach. To do so, we modified Linux's existing kernel-based implementation of NFS version 2. (We used version 2 as a base because version 3 is not available as a loadable module, hindering debugging.) NFS and its utilities comprise about 45,000 lines of C. To this we added: (1) new filesystem code comprising 7,500 lines (4,000 at the server and 3,000 at the client); (2) new disk functionality comprising about 1,000 lines (see Figure 5); and (3) a security library of about 14,000 lines, the vast majority of which was imported from openssl.

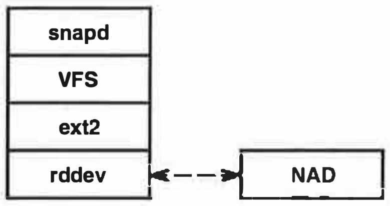


Figure 6: **Snapdragon server and NAD.** When the Snapdragon server (snapd) receives a client request, it passes it to Linux's VFS, which invokes the underlying filesystem (ext2 in our case), which in turn issues block requests to the device driver rddev. The latter translates these requests to NAD requests containing "allow-all" capabilities.

3.1 Overview

Snapdragon clients run two kernel modules: a standard NFS lock daemon and a module that contains the core filesystem functionality, including requesting and caching capabilities and (partial) blockmaps. The second module exports through Linux's Virtual File System (VFS) interface the new filesystem type "snapfs".

The Snapdragon metadata server consists of a filesystem kernel module (snapd), a device driver (rddev) and a lock daemon (lockd). Lockd is identical to NFS's. Snapd and rddev are shown in Figure 6. Snapd translates client requests into the filesystem-independent operations. In Linux such operations are handled by the VFS layer, which invokes filesystem-specific code (in our case ext2) that implements the operation by issuing low-level block requests to the device driver, rddev in this case. Rddev translates these block requests to messages to the disk controller (NAD) using the same protocol as the clients use, but using "allow-all" capabilities. This architecture allows Snapdragon to be independent of the underlying file system and allows the data layout on remote disks to be exactly the same as if the disks were local. This has nice implications for deployment as we explain in Section 4.3.

The Snapdragon disk controller is implemented as a PC connected to the network. The PC runs a small multi-threaded user-level program that listens for, checks, and executes block requests.

3.2 Changes to the NFS protocol

The Snapdragon metadata server implements a superset of the NFS protocol. Snapdragon clients do not issue the NFSREAD and NFSWRITE RPCs to the metadata server, because reading and writing are handled locally at the client by issuing block read and write requests directly to the relevant NAD based on cached blockmaps and capabilities. Unmodified legacy NFS clients can continue to talk to a Snapdragon metadata server using standard NFS RPCs. Metadata consistency is ensured because all metadata commands are still handled by the server.

Three commands were added to the NFS protocol: OPEN, CLOSE, and GETCAPS. When issuing a GETCAPS call, the client passes a file handle, an access mode, and a range of logical blocks in the file, presumably because it would like to read or write these blocks in the future. The metadata server returns a blockmap specifying the corresponding physical blocks and capability(s) giving the client the requested access to the requested blocks. To keep messages within a reasonable size, if the range of requested blocks is very large or fragmented, the server returns a blockmap and set of capabilities which cover a range as large as possible, and the client must send another request for the remaining blocks.

Because GETCAPS returns a blockmap as well as capabilities, we can invalidate a file's blockmap by revoking the file's capabilities: any attempt by a client to use the old blockmap will result in an revoked-capability error from the disk, forcing the client to do a GETCAPS before it can proceed, giving it the new blockmap.

The OPEN and CLOSE commands are necessary only for caching file contents at the client, not for security. In our system, every open file is in either *exclusive* or *non-exclusive* mode. A file is in exclusive mode if either precisely one client has it open (reading or writing) or no client has it open for writing. When a client has a file open that is in exclusive mode, it knows that the file cannot undergo changes it is not aware of, and therefore it can cache the file's contents. In other situations, clients must use short timeouts on their cache in order to achieve acceptable levels of consistency. (We provide the same consistency as NFS.[4]) Because the server is notified whenever a client opens or closes a file, it knows when the exclusivity of a file changes, and can notify the clients that have that file open by using callbacks.

[4] A higher level of consistency could be implemented using the same basic technique as Spritely NFS [24]: channeling reads and writes for non-exclusive-mode files through the metadata server.

We take advantage of the need for an OPEN call, by piggybacking (on the reply to the client) a blockmap and capabilities for the opened file covering as many blocks as possible. This dramatically reduces the need for GETCAPS calls.

One might think that the separation of metadata and data in a NAD file system would require appending to a file be given special treatment. It turns out that append operations can be subsumed under standard write operations, by using Unix's *bmap* function with the *allocate* flag set. This function maps a logical block of a file to a physical block. If the block is not yet mapped and the allocate flag is set, the block is allocated according to the specifics of the underlying file system. Thus, to append to a file, a client issues a standard GETCAPS call for write access to logical blocks beyond the current end of the file. The metadata server can then simply call `bmap` to simultaneously allocate and get the newly appended block.

3.3 Capabilities

The design of our capability format was guided by studying the properties of the AdvFS file systems at our 30-person research laboratory. Recall that a capability includes a fixed-size list of extents, which are contiguous ranges of physical blocks. Files occupy one or more extents; for example, `/foo/bar` might occupy blocks [243-256], [9323-9992], and [20-50]. In our file systems, a single extent covers, on average, 150 KB. Moreover, it turns out that 90% of files require four or fewer extents, and that 95% require 13 or fewer extents. Hence, we decided that any single capability would have space allocated for four extents. Files with more than four extents can be accessed with multiple capabilities.

3.4 Bloom filter parameters

We use two 32KB bloom filters (262,144 bits each). We determined the other parameters by optimizing, using statistical simulation, for the maximum number of requests on average that can be supported per epoch subject to a maximum false-positive rate, measured over the last 1,000 requests, of 0.1%. The resulting parameters—$n = 9$ hash functions and about 47% of bits used in a full filter—yield epochs lasting 18,640 requests on average, or 30 minutes under the request rate of the trace used in Section 4.1.

3.5 Cryptographic details

Wherever a MAC is required, we use HMAC with the Secure Hash Algorithm SHA-1 [17]. This function returns 20-byte hashes, can be computed extremely fast, and possesses no known collisions. The client/server secure channel (see Figure 2) is achieved using the Blowfish block cipher algorithm [23] with a 16-byte key. When privacy is desired, we use DES encryption on messages to and from the disk.

Key management is rudimentary in the current prototype: all keys are read from configuration files and remain fixed indefinitely. Naturally, in a mature system, one could use a more elaborate scheme like the one described by Gobioff *et al.* [10].

3.6 Packet security overhead

Capabilities occupy 72 bytes using generous 64-bit values for block numbers. The replay epoch number has 64 bits, the nonce plus random padding for encryption use 128 bits, and the MAC has 160 bits. Thus, the total security data in a disk request is 116 bytes compared to up to 8192 bytes of payload.

4 Discussion

4.1 Practical benefit of capability groups

Is the capability group method beneficial in practice? In particular, do capability groups reduce the burstiness of capability allocations, thus reducing the chance of the metadata server being overloaded? To answer this question, we simulated the behavior of secure NADs using the trace of a 500 GB file system used by about 20 researchers over 10 days [21]. The results are shown in Figure 7, which is a histogram of how many capability allocations were performed by the metadata server in each 1-second period. The amount of memory for capability storage was fixed at 64 KB. The black bars show results for the straightforward method of Section 2.2, which uses a single bitmap to store all capabilities; this is precisely equivalent to the capability group method, with the number of groups $g = 1$. The white bars are for a configuration using the same amount of memory, but divided into $g = 64$ groups.

Note that for this particular workload, the metadata server could hardly be described as "overloaded": the peak load of around 500 capabilities/second can easily

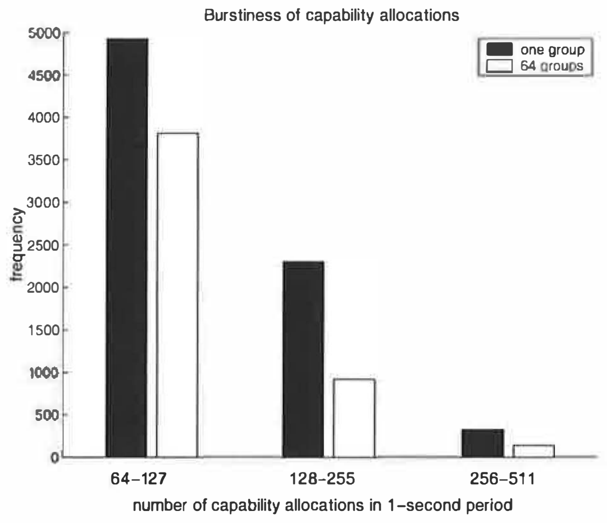


Figure 7: **Using 64 groups instead of 1 substantially reduces the burstiness of capability allocations.**

be handled since it takes less than 2 ms to issue a capability. Nevertheless, these results demonstrate that the capability group approach substantially reduces burstiness. If the stress on the server were greater (*e.g.*, if it served more disks), the capability group approach would improve performance by reducing periods of overload.

4.2 Choice of underlying file system

Our approach to security can be achieved by incrementally modifying most types of file systems. The major exceptions are file systems that store data from multiple files in the same block, such as ReiserFS [20] and FFS [14], which store the tails of multiple files in a single block. These file systems cannot be used because they require clients be given access to only part of a block, which our block-based capabilities cannot handle.

4.3 Deployment

Our approach is easy to deploy incrementally in existing legacy NFS environments. So long as the same underlying file system (and hence, disk layout) is kept, we could take existing disks with data and use them (without reformatting) as NADs by attaching them to the network via a controller that checks for capabilities and replay attacks. (Of course, this controller would have to be specially manufactured for use with our protocol.) By having our metadata server also export the Snapdragon file system via plain NFS, we can support legacy NFS clients, albeit with poorer performance. This allows an NFS system to be converted one client at a time.

4.4 Optional improvements to capability groups

There are some improvements that could be applied to our basic technique of capability groups, which we now describe for completeness. However, for practical purposes we found that Snapdragon performed quite well even without these optimizations—in fact, our performance numbers in Section 5 do not include them.

Note that when a group is invalidated, there will be some *unintended revocations*, that is, valid capabilities will be revoked even though the permissions of their files have never changed. This of course does not break the correctness of the protocol: the client with an invalid capability can simply request a new capability from the metadata server. However, performance is affected because this procedure costs two extra network round trips: one when the client, unaware, attempts to use the invalid capability and gets rejected, and another round trip when the client requests the new capability.

It is desirable to minimize the number of such unintended revocations. An obvious strategy is to choose for invalidation a group with many revoked capabilities and few valid ones. In addition, one can exploit the fact that different capabilities have different (probabilistic) lifetimes. For example, a read-only capability for a shared library is unlikely ever to be revoked, whereas a read/write capability for a recently-created private file in `/tmp` is likely to have a much shorter lifetime. Thus, we could designate certain groups as volatile and others as stable, possibly with gradations in between, and assign capabilities to appropriate groups. Volatile groups would then become good candidates for low-cost invalidations.

Another way to minimize unintended revocations is to do background cleaning. When the metadata server is idle, it can help increase the number of available capability ID's by choosing one or more capability groups—preferably groups with many entries in the revocation list—and slowly migrating their valid capabilities to other groups. Migrating a capability means issuing an equivalent replacement capability in a different group and giving it out to clients that have the old capability; the clients replace the old capability with the new one. Once all valid capabilities in a group have been migrated, the metadata server can invalidate that group without causing any unintended revocations. Note that this scheme requires the metadata server to issue callbacks to the clients.

5 Performance

We ran experiments to evaluate the following: (1) the overhead of security, including MAC computation, capability revocation, and encryption; and, (2) system throughput and scalability under a bandwidth-intensive workload. Since the motivation of this work is to extend the performance benefits of NAD file systems to insecure environments, it is essential that the performance advantages of NAD file systems not be significantly reduced when security is added. We repeated each experiment using several different setups for comparison purposes.

The setups we used include the following: *non secure*, Snapdragon with all security turned off; *secure*, Snapdragon with access control, but without encryption; *private*, Snapdragon with access control and encryption; and, *NFS*, an NFS server with an attached local disk. Access control refers to the capability operations and replay detection needed to prevent unauthorized operations. Except where otherwise noted, encryption in this section refers to the encryption of all messages to and from the disk for privacy, and not to the encryption used for the client/server channel, which is part of Snapdragon's access control and hence present in both the secure and private setups. The non-secure setup does no MAC calculations, replay detection, capability operations, or encryption of any kind.

Our experiments were conducted on 3 to 8 Celeron 400 MHz PC's running Linux kernel version 2.4.12 and connected with a gigabit Ethernet switch. "Jumbo" 9,000-byte frames were enabled for network communications. Each machine has a locally-attached IDE disk with a maximum bandwidth of approximately 25 MB/second. In each experiment, one machine acts as the diskless metadata server, while others act as simulated disk controllers or diskless clients. (A simulated disk controller is the user-level program described in Section 3.1, which uses a raw disk partition as its backing store.)

A major difference between a real hardware NAD and our simulated one lies in the amount of memory available for the data cache. A commodity disk drive typically has a few megabytes, while the machines hosting our simulated NAD have 128 MB. Such a large cache would have a significant impact on NAD performance, because the disk controller could buffer and coalesce small random accesses into large sequential ones, improving the utilization of raw disk bandwidth.

Therefore, in order to make our simulated disk controllers more realistic, we limit their cache to 2 MB for these setups; that is, we force a sync to disk for every 2 MB of dirty data that a simulated NAD receives; such scheme is appropriate for the streaming performance tests that we ran. In addition, we took the following measures to minimize the unintended effects of buffer caches: we freshly mounted the file systems and invalidated all block-device buffer caches before each experiment started, and flushed all buffer caches and unmounted the file systems before each experiment completed.

The capability scheme used in the experiments is the capability group method as described in Sections 2.2 and 3.4. But with the parameter values suggested there, group invalidations are very rare. To ensure the experiments included any performance implications of group invalidations, we used a much smaller store of capabilities—a strictly pessimistic alteration. Specifically, we set the number of groups (g) to 20, and the maximum number of capabilities in each group (w_B) to 500, allowing a maximum of 10,000 allocated capabilities. Therefore, for every 500 capabilities allocated beyond the first 10,000, a group needed to be invalidated.

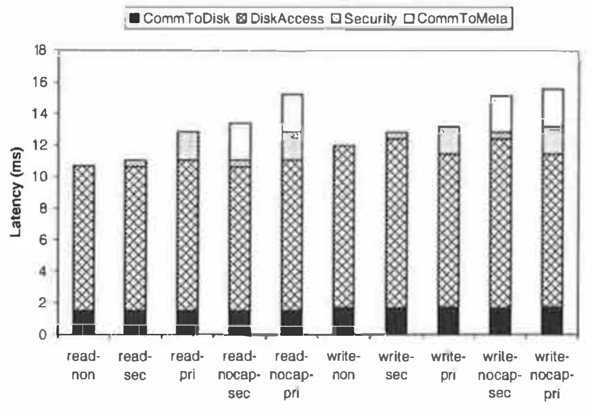


Figure 8: **Latency breakdown of read and write operations** for non-secure setup (non), secure setup (sec), and private setup (pri). Operations labeled with "nocap" means that the client does not have the appropriate capability and thus it has to request one to the metadata server. Latency is divided into the following categories: communication to the metadata server (CommToMeta), communication to the disk controller (CommToDisk), disk access, and security (including MAC computation and encryption).

5.1 Latency breakdown

We ran a set of micro benchmarks on Snapdragon and measured the latency of each operation in order to evaluate the various overheads associated with our security scheme. All the latency benchmarks were run on a collection of 700 files, each of size 4 KB. In each benchmark, a fixed filesystem operation (*e.g.*, read or chmod) was performed on each of the files in a randomized order. For the read and write cases, the metadata server, simulated NAD, and client driver were instrumented to report the time spent in fine-grained sub-operations.

Figure 8 shows the latency breakdown of the read and write operations with empty and synchronous write-through caches respectively. The physical disk access time averaged 9.3 ms for reads and 10.2 ms for writes. The MAC computation overhead was 0.4 ms and the encryption overhead was 1.4 ms. The disk-communication latency for all operations was 1.6 ms. If the client needs to request a capability for an operation, it requires an additional round trip from the client to the metadata server, which costs 2.3 ms. If a client attempts to use a revoked capability (not shown), it will get a rejection from the disk, which costs an extra 1.8 ms (secure setup) or 2.5 ms (private setup).

Figure 9 shows the latency of metadata operations. The chmod operation involves a round trip from the metadata server to the simulated NAD that requires MAC computation, while the unlink operation involves multiple such trips because ext2's unlink code writes multiple disk blocks. The open operation involves both MAC computation (to compute the secret s) and encryption of the capability, whether or not encryption for privacy is used. For operations that do not require revocations, the overhead for access control is less than 1 ms and the overhead for privacy is less than 3 ms. The operations involving revocations (*i.e.*, chmod-rev and unlink-rev) require an additional round-trip from the metadata server to the disk, roughly 1.4 ms.

In summary, access control (*i.e.*, MAC computation and replay detection) increases the latency of reads and writes by less than 0.5 ms (5%); encryption an additional overhead of 1.4 ms; and capability revocation increases the latency of read or writes latency by roughly 2.3 ms. For metadata operations, access control costs can cost 1 ms and privacy can cost 3 ms for certain operations.

5.2 Aggregate throughput and scalability

We ran a benchmark to measure the bandwidth of reads and writes by multiple clients on a single disk with various file sizes. There were 6 clients in our experiments, each running on a separate machine. Each client opened one file at a time and sequentially read or wrote in 64 KB chunks. Each experiment lasted between 5 and 25 min-

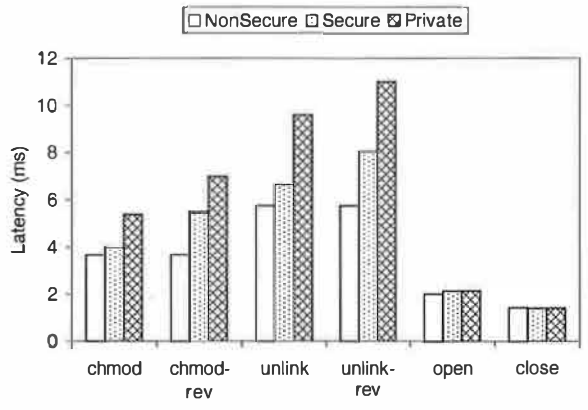


Figure 9: **Latency of metadata operations** under the non-secure, secure, and private setups. Operations labelled with "rev" require a capability revocation message to be sent from the metadata server to the disk.

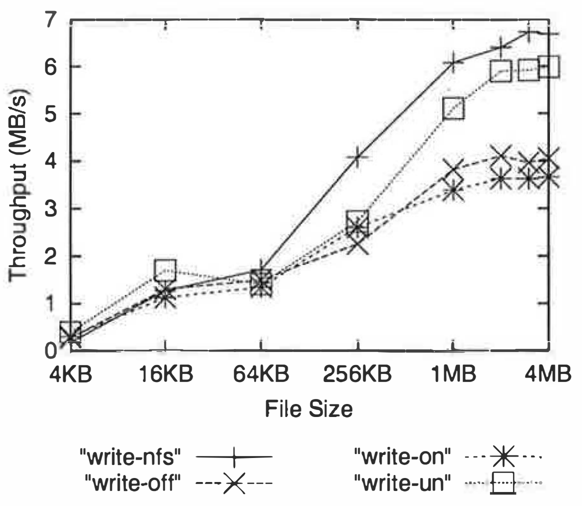


Figure 10: **Aggregate write bandwidth with 1 disk** and 6 clients for the secure setup (on), the non-secure setup (off), the NFS setup, which has no cache limit (nfs), and the non-secure setup modified to have no cache limit (un).

utes. All the files were stored on the same disk, but no file was accessed by more than one client or more than once during each experiment. All files within an experiment have the same size, but size varies from 4 KB to 4 MB between experiments.

The larger the file size, the less open/close overhead is incurred per transferred byte. There is also overhead associated with capability-group invalidation; the benchmarks using file sizes of 4 KB and 16 KB accessed more than 10,000 files and hence triggered group invalidation.

Figure 10 shows the system throughput as a function of file size for the write benchmarks. (The read benchmark results have similar trends and are not shown.) With file size 256 KB or less, the secure and non-secure setups have comparable bandwidth. With file sizes larger than 256 KB, the secure system performs up to 16% worse than the non-secure system. The difference is caused by CPU contention on the disk machine. Figure 11 shows the average percentage of idle time on the machine where the simulated NAD was hosted. The simulated disk controller in the secure setup consumes a considerable amount of cycles for MAC computation. Since it is implemented as a user-level process, it also consumes cycles for context switching and moving data across PCI buses and the kernel boundary.

We ran the same benchmark on an NFS server with a locally-attached disk (the NFS setup) for comparison. NFS performs comparably to Snapdragon (secure and non secure) for file sizes of 64 KB or less, and noticeably better for file sizes larger than 64 KB. This better performance is due to the NFS server's large data cache. Therefore, we ran the same benchmark again using the non-secure setup modified so that the simulated NADs can use as much cache as possible, up to the 128 MB physical memory capacity. The result is shown in Figures 10 and 11 as "write-un". The non-secure setup with no cache limit performed significantly better than the standard non-secure setup, which has only 2 MB of cache. This suggests that it would be worth increasing the data cache capacity in NADs (secure or non secure) in order to maximize bandwidth utilization for streaming I/O of large files by many concurrent users.

The idle time of the NFS server (shown in Figure 11) is not monotonic because the NFS server is performing both metadata and I/O operations. As the file size increases, the rate of metadata operations decreases, but the I/O rate increases.

We also ran a benchmark to measure the aggregate throughput for various numbers of disks and clients. Due to the limited number of machines available for our experiments, we had to collocate a client with a simulated NAD controller on each machine. Each client sequentially read or wrote files on a NAD hosted by another machine and each NAD was accessed by exactly 1 client. We ran the benchmark with 2 through 7 such machines. The file size was 256 KB and each client accessed 600 files in each run. Figure 12 shows the aggregate bandwidth as a function of the number of disks.

The results show that the aggregate read or write bandwidth of all clients scales linearly with the number of

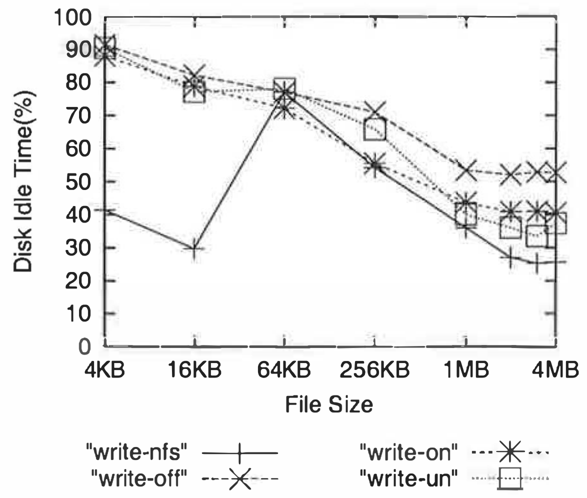


Figure 11: **Average percentage of idle CPU time on disk machines.**

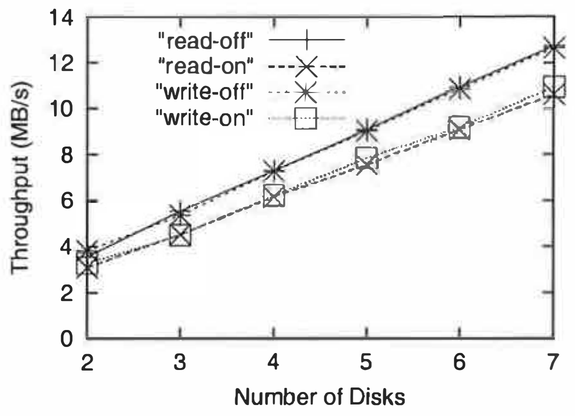


Figure 12: **Aggregate read/write bandwidth with multiple disks** for secure setup (on) and non-secure setup (off).

disks, which indicates that the metadata server imposes very low overhead to a high-bandwidth workload and has not become a bottleneck in a system with up to 7 disks. Figure 13 shows the average percentage of idle CPU time on the metadata server machine. The metadata-server machine was underloaded (*i.e.*, 86-92% idle) in these experiments. Therefore, we expect it to be able to support a considerably larger number of disks. The throughput of the non-secure setup grew faster than that of the secure setup because the access control overhead, which is dominated by MAC computation, is proportional to the data bandwidth.

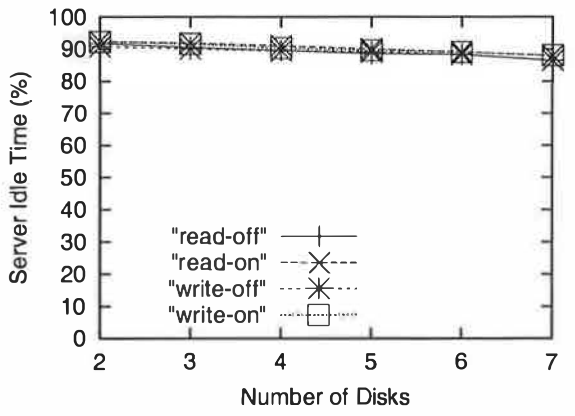


Figure 13: **Average percentage of idle CPU time on the metadata server.**

5.3 Andrew benchmark with Linux kernel source

We ran a variant of the Andrew benchmark to show that Snapdragon has acceptable performance on a standard benchmark, even though Snapdragon was not designed for such workloads (*e.g.*, with extensive metadata operations and small files). Our variant of the Andrew benchmark differs only in that it uses as input the Linux kernel source, which contains 690 directories, 10,528 files and roughly 127 MB of data. Phase I of the Andrew benchmark duplicates the 690 directories 5 times in the file system being tested; phase II copies the files into one of the duplicated directories; phase III recursively lists all the duplicated directories; phase IV scans each copied file twice; and, phase V does a "make dep" and then "make" in the copied Linux kernel source directory, generating 1,362 new dependency and object files, or 13 MB of data.

The configuration for the Andrew benchmark includes only one client and NAD (or NFS server), each using separate machines. Figure 14 shows the elapsed time for each phase of Andrew benchmark for the secure, non-secure, and NFS setups.

In all phases, Snapdragon performed almost the same whether security was turned on or off, suggesting that the overhead of security is low. Snapdragon and NFS differed somewhat in phases I and II. The difference in phase I occurs because, for each new directory to be created, the Snapdragon metadata server needs to access the disk across the network, while the NFS server accesses the local disk directly. The difference in phase II is due to the overhead in opening and closing small files in Snapdragon—the Linux kernel source consists of mostly small files: 97% of the files are less than 64 KB,

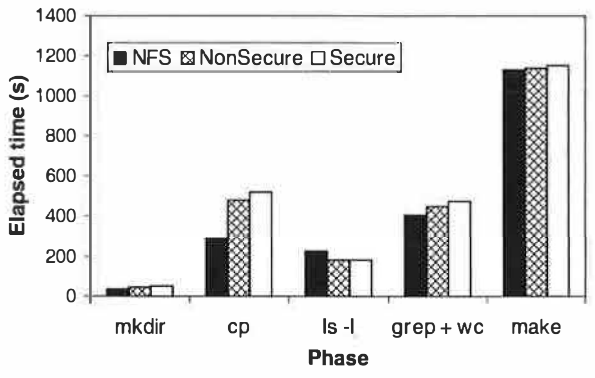


Figure 14: **Andrew benchmark with Linux kernel source.**

all but one file is less than 900 KB, and the largest file is roughly 2 MB. In phases III, IV and V, NFS and Snapdragon performed almost the same.

6 Limitations

One limitation of our block-based security approach is that we do not support files that are writable but not readable. This is because, under our scheme, to execute partial writes (writes to only part of a block), the client first needs to read the block's old contents—which requires read access—so that he can modify it. However, it is possible to overcome this problem by either having the disks support partial writes directly, or by having all such writes go through the metadata server.

We also do not support underlying file systems in which a block can contain data belonging to multiple files (*e.g.*, file systems in which the tails of many files are stored in a single block), because a block is the smallest unit of access control in our scheme. However, this problem can be overcome by changing capabilities so that they can optionally restrict access to a range of bytes within a block and by allowing disks to accept partial read and write requests.

Finally, with a log-based file system, it is not easy to exploit direct client access to disk when writing to the log, because accesses to the log need to be serialized. (This is not, however, a drawback of our security scheme, but rather a general limitation of asymmetric shared file systems with network-attached disks.) One possible solution is to make the disk the serialization point, but doing so would require adding considerable functionality to the disk.

7 Related work

NASD [7, 8, 10, 9] introduced the basic security architecture we use, where a central server decides policy and the disks implement only a simple access mechanism based on cryptographic capabilities. We differ in our handling of revocations and replay attacks, as described in Sections 2.2 and 2.4, and in the fact that our capabilities specify permission in terms of ranges of physical blocks rather than object IDs. SCARED [19] generalizes the NASD security framework to allow for identity-based access, where the client proves its identity to the disk, which then decides what access should be allowed based on its understanding of file access-control lists (ACLs). We prefer a capability-based scheme, however, because it does not require restrictions on the disk layout so that the disk can decode ACLs.

NASD's network-attached disks are complicated pieces of machinery, possessing most of the functionality of a file server: rather than simply serving raw blocks as our NADs do, they present the abstraction of a collection of numbered, but unnamed, variable-size data objects, which are byte sequences with a small number of attributes. A central server manages a collection of NASD disks, providing clients with the usual illusion of a hierarchical file system. This is done (in the absence of striping) by mapping each directory or data file to a single NASD object.

SUNDR [13] and SNAD [5, 15] do away with the central server altogether and use more powerful cryptographic methods (*e.g.*, blocks on disk are encrypted and "signed" using keys known only to clients) to stop an attacker that can control disks from reading or undetectably altering user data. Most of their complexity is due to this stronger security level, which is not useful for the scenarios that we envision, where the people most likely to be able to compromise servers or disks (*i.e.*, the system administrators) would also easily be able to compromise clients, defeating these systems' security.

SUNDR and SNAD use the *encrypt-on-disk* strategy, where data is concealed by keeping it encrypted on the disk. Revoking access in such systems is expensive because the involved data must be re-encrypted using a new key. Riedel *et al.* [21] argue that encrypt-on-disk may offer better performance when privacy is desired than *encrypt-on-wire* systems such as ours, because encrypt-on-disk encrypts and decrypts data only at the client, not at both the client and disk.

Although encrypt-on-disk leaks more information to eavesdroppers (blocks can be identified on the wire because they are re-encrypted only when rewritten, instead of each time they are transmitted), should the extra encryption prove burdensome we think it is possible to modify Snapdragon to get the performance benefit (but not the security benefit) of encrypt-on-disk without changing our disk protocol.[5]

The Netstation project sketches how their *Derived Virtual Device* (DVD) abstraction, a very general mechanism for securely delegating access to an arbitrary subset of a network-attached device, can be used to create a block-oriented secure NAD file system called STORM [26]. Rather than use capabilities for security, they use Kerberos authentication to authenticate client requests, which specify a DVD ID. Devices maintain a table of which DVDs each client is allowed to access as well as detailed information about the access restrictions of each DVD. While the notion of DVDs is very elegant, it may be too inefficient to be of use in a production file system: the DVD access information (largely blockmap information for STORM) requires extra network trips to be installed by the server, uses a lot of disk memory, and seems to be installed by downloading functions written in Scheme.

We believe our paper is the first one to use Bloom filters to protect against replay attacks. However, Bloom filters have long existed and have other uses. Superficially related to our paper is the work of [6], which uses Bloom filters for revocations (instead of replay attacks).

8 Conclusion

In this paper we have presented a new block-based security scheme for network-attached disks (NADs). In contrast to previous work, our scheme requires no changes to the data layout on disk and only minor changes to the standard protocol for accessing remote block-based devices. Thus, existing NAD file systems and storage-management software could incorporate our new secure NADs with only incremental changes. Moreover, our scheme's demands on the NADs are modest: standard cryptographic functionality plus very little RAM. The low need for RAM is achieved by two novel features: our revocation scheme based on capability groups, and a replay-detection method using Bloom filters. We believe our design could be easily deployed in existing NAD's or in disk arrays with minimal changes.

We implemented a prototype secure NAD file system using our scheme, and evaluated its performance and scalability. The cost of access control is small: Latency for reads and writes increases by less than 0.5 ms (5%), and the bandwidth decreases by up to 16%. The system throughput scales linearly with the number of disks supported by a single metadata server (up to 7 in our experiments).

Hence, we believe our scheme is a practical and efficient method for incorporating security into existing NADs with minimal change—a scheme that could liberate NAD file systems from the confines of the machine room and data center, allowing them to reach a broader range of users directly, yet securely.

[5] We would store blocks encrypted on disk, but the keys would be managed by the metadata servers.

Acknowledgments

We would like to thank Fay Chang and John Wilkes for useful discussions as well as our shepherd Greg Ganger and the anonymous referees for helpful suggestions.

References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *Lecture Notes in CS*, 1109:1–15, 1996.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] Andrew D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.

[4] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970.

[5] W. Freeman and E. Miller. Design for a decentralized security system for network-attached storage. In *Proceedings of the 17th IEEE Symposium on Mass Storage Systems and Technologies*, pages 361–373, March 2000.

[6] Eran Gabber and Abraham Silberschatz. Agora: A minimal distributed protocol for electronic commerce. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, pages 223–232, 1996.

[7] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks. Technical Report CMU–CS–97–112, Carnegie Mellon, March 1997.

[8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, Fay W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. A. Zelenka. Cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 92–103, October 1998.

[9] Howard Gobioff. *Security for a High Performance Commodity Storage Subsystem.* PhD thesis, CMU, 1999.

[10] Howard Gobioff, Garth Gibson, and Doug Tygar. Security for network attached storage devices. Technical Report CMU–CS–97–185, Carnegie Mellon, October 1997.

[11] Kent Koeninger. CXFS: A clustered SAN filesystem from SGI. http://www.sgi.com/Products/PDF/2691.pdf.

[12] Barbara Liskov, Liuba Shrira, and John Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.

[13] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 99–104, May 2001.

[14] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for UNIX. *ACM Trans. on Computer Systems*, 2(3):181–197, 1984.

[15] Ethan L. Miller, William E. Freeman, Darrell D. E. Long, and Benjamin C. Reed. Strong security for network-attached storage. In *Proceedings of the FAST 2002 Conference on File And Storage Technologies*, pages 1–14, January 2002.

[16] S. Mittra and T. Woo. A flow-based approach to datagram security. In *Proc. ACM SIGCOMM*, 1997.

[17] NIST. Secure hash algorithm, 1995. FIPS 180-1.

[18] Matthew T. O'Keefe. Shared file systems and Fibre Channel. In *Proceedings of the Sixth NASA Goddard Conference on Mass Storage Systems*, pages 1–16. IEEE Computer Society Press, 1998.

[19] B. Reed, E. Chron, D. Long, and R. Burns. Authenticating network attached storage. *IEEE Micro*, 20(1), January 2000.

[20] Hans Reiser. Reiserfs v.3 whitepaper, 2000. http://www.namesys.com/.

[21] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)*, pages 15–30, January 2002.

[22] R. Sandber, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of USENIX Summer Conference*, 1985.

[23] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[24] V. Srinivasan and J. C. Mogul. Spritely NFS: Experiements with cache-consistency protocols. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 45–57, December 1989.

[25] Tivoli. Tivoli SANergy: Helping you reach your full SAN potential. http://www.tivoli.com/-products/documents/datasheets/sanergy_ds.pdf.

[26] Rodney Van Meter, Gregory Finn, and Steven Hotz. Derived virtual devices: A secure distributed file system mechanism. In Ben Kobler, editor, *Proc. Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, September 1996.

The Direct Access File System

Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent
Dave Noveck, Tom Talpey, Mark Wittle
Network Appliance, Inc.

Abstract

The Direct Access File System (DAFS) is a new, fast, and lightweight remote file system protocol. DAFS targets the data center by addressing the performance and functional needs of clusters of application servers. We call this the *local file sharing* environment. File access performance is improved by utilizing Direct Access Transports, such as InfiniBand, Remote Direct Data Placement, and the Virtual Interface Architecture. DAFS also enhances file sharing semantics compared to prior network file system protocols. Applications using DAFS through a user-space I/O library can bypass operating system overhead, further improving performance. We present performance measurements of an IP-based DAFS network, demonstrating the DAFS protocol's lower client CPU requirements over commodity Gigabit Ethernet. We also provide the first multiprocessor scaling results for a well-known application (GNU gzip) converted to use DAFS.

1 Introduction

With the advent of the Virtual Interface Architecture [11], InfiniBand [1], and the Remote Direct Data Placement (RDDP) protocol [13], messaging transports that support Remote Direct Memory Access (RDMA) operations are moving from the realm of esoteric, single-vendor implementations into the mainstream of commodity technology. The DAFS Collaborative was launched in 2000 with the mission of developing a network file access protocol, the Direct Access File System (DAFS) [9], that takes full advantage of such *Direct Access Transports* (DATs), building on the semantics of existing network file system protocols. DAFS employs many concepts and semantics of NFSv4 [12], which in turn is derived from earlier versions of NFS and incorporates some semantic features of CIFS [30]. Unlike NFSv4, DAFS includes semantics for which there were no pre-existing APIs. The Collaborative defined the DAFS API to access them.

The design of DAFS targets two goals. The first is to enable low-latency, high-throughput, and low-overhead data movement between file system clients and servers. Part of this goal is to minimize the demands placed on the client CPU by the protocol. DAFS addresses network throughput and latency concerns by using fast DAT technologies built upon RDMA capability. It minimizes client CPU utilization by using the direct data placement and operating system bypass capabilities of the transports and by placing a premium on client CPU cycles at all stages of the protocol. By minimizing client CPU utilization, DAFS clients can potentially experience better performance accessing files via DAFS than they can by accessing files through their local file systems.

The second goal of DAFS is to include the necessary semantics to ensure reliable, shared access in a clustered system. Because it is targeted at highly scalable workloads within data center environments, support for clustered clients is essential. DAFS contains many features that enable sharing of files among clients, while allowing cooperating clients to limit access to shared files by outsiders.

1.1 Motivation

Files can be accessed in three ways: from local disk, from a remote shared disk, and over a network file system. Traditionally, local file systems [26, 28] accessing local disk storage have provided the highest performance access to file resident data. However, they do not solve the problem of sharing data sets or storage among a number of computers. Cluster file systems, such as Frangipani [20, 34], GPFS [29], and GFS [32], allow multiple clients to access a single remote shared disk pool, by coordinating management and control of the shared file system among the clients. These systems access shared disks using transports such as iSCSI and Fibre Channel [31] that are designed for data storage, so they enjoy bulk data transfer performance on par with local disk. The price paid is relatively complex distributed data and metadata locking protocols, and complex failure handling. These systems also demand client software homogeneity.

Network file systems [16, 27] permit heterogenous clients to share data on remote disks using more traditional networks. This characteristic comes at the expense of relatively poor performance compared to local file systems, due in part to the high CPU and latency overhead of the network protocol layers, and the limited bandwidth of the transports. Network file systems avoid much of the complexity of cluster file system implementations by using messaging protocols to initiate operations that are performed on the server. The implementation of data sharing, fault isolation, and fault tolerance is simplified compared to cluster file systems because data and metadata modifying operations are localized on the server. Network file systems are typically less tightly integrated than cluster file systems, requiring a lower degree of locality and cooperation among the clients. We believe that there is a role for a high-performance network file system in tightly coupled environments, because of the inherent advantages of simple data sharing, interoperable interfaces, and straightforward fault isolation.

Based upon almost two decades of experience with the network file system model, it was apparent to the participants in the DAFS Collaborative that a new network file system protocol based on RDMA-capable Direct Access Transports would be most useful as a step beyond NFSv4. This need was recognized before NFSv4 was finalized, but after NFSv4 was largely defined. Unlike NFSv4, DAFS was targeted specifically at the data center, incorporating what was called the *local sharing* model as its design point. Hence DAFS assumes a higher degree of locality among the clients than NFSv4, and is designed to enable a greater degree of cooperation among those clients and the applications running on them. The intent is to enhance the support for clustered applications, while maintaining the simple fault isolation model characteristics of the network file system model.

One design alternative would be to leverage the existing ONC RPC [33] framework that underlies NFS by converting it to use RDMA for data transfer [18]. This approach would enable direct data placement, but the existing framework would not allow several other optimizations, including full asynchrony, session-oriented authentication, request throttling, and resource control. While it is likely that a reasonable modification to ONC RPC to accommodate these goals could be done, it would involve fairly major changes in the RPC framework and interface. Our goal was high-performance file access, not a general purpose RPC mechanism.

A goal of DAFS is to maximize the benefit of using a DAT. To accomplish this, it was not sufficient to simply replace the transport underneath the existing RPC based NFS protocol with an RDMA-aware RPC layer. DAFS enables clients to negotiate data transfers to and from servers without involving the client CPU in the actual transfer of data, and minimizes the client CPU required for protocol overhead. DATs offload much of the low-level network protocol processing onto the Network Interface Card (NIC), and the interfaces to these network adapters, such as DAPL [10], are callable from user space and do not require transitions into the kernel. Thus, it is possible to reduce the client CPU load of the network file system to just the costs of marshalling parameters for remote operations, and unpacking and interpreting the results. Data can be placed directly in the applications' memory by the NIC without additional data copies. DAFS further reduces the overhead of data transfer by incorporating batch and list I/O capabilities in the protocol.

To support clustered clients, DAFS extends the semantics of NFSv4 in the areas of locking, fencing, and shared key reservations. It does not require that the cluster file system implementations built using DAFS fully support POSIX file access semantics, but does provide sufficient capability to implement such semantics if desired.

The remainder of the paper is as follows. Section 2 provides some background on Direct Access Transports, the network technology required for DAFS. Section 3 introduces the core of the Direct Access File System, focusing on the rationale behind significant design decisions. Section 4 follows with the DAFS API, the standard interface to a user-space DAFS client. Section 5 demonstrates the suitability of DAFS for local file sharing by providing some performance results. Finally, the paper concludes with a summary of our work.

2 Direct Access Transports

Direct Access Transports are the state of the art in faster data transport technology. The high-performance computing community has long used memory-to-memory interconnects (MMIs) that reduce or eliminate operating system overhead and permit direct data placement [3, 4, 5, 15]. The Virtual Interface Architecture standard in the mid 1990's was the first to separate the MMI properties and an API for accessing them from the underlying physical transport. Currently, InfiniBand and RDDP are positioned as standard commodity transports going forward.

The DAT Collaborative has defined a set of requirements for Direct Access Transports, as well as the *Direct Access Provider Layer* (DAPL) API as a standard programming interface [10] to them. The DAT standard abstracts the common capabilities that MMI networks provide from their physical layer, which may differ across implementations. These capabilities include RDMA, kernel bypass, asynchronous interfaces, and memory registration. Direct memory-to-memory data transfer operations between remote nodes (RDMA) allow bulk data to bypass the normal protocol processing and to be transferred directly between application buffers on the communicating nodes, avoiding intermediate buffering and copying. Direct application access to transport-level resources (often referred to as *kernel bypass*) allows data transfer operations to be queued to network interfaces without intermediate operating system involvement. Asynchronous operations allow efficient pipelining of requests without additional threads or processes. Memory registration facilities specify how DAT NICs are granted access to host memory regions to be used as destinations of RDMA operations.

DAPL specifies a messaging protocol between established endpoints, and requires that the receiving end of a connection post pre-allocated buffer space for message reception. This model is quite similar to traditional network message flow. DATs also preserve message ordering on a given connection. The DAFS request-response protocol uses DAPL messaging primitives to post requests from client to server and receive their responses on the client. DAFS also uses RDMA to transmit bulk data directly into and out of registered application buffers in client memory. RDMA can proceed in both directions: an RDMA *write* allows host A to transfer data in a local buffer to previously exported addresses in host B's memory, while an RDMA *read* allows A to transfer data from previously exported addresses in B's memory into a local buffer. In the DAFS protocol, any RDMA operations are initiated by the server. This convention means that a client file *read* may be accomplished using RDMA *writes*. Conversely, a client may write to a file by instructing the server to issue RDMA reads.

Any storage system built using a DAT network can be measured in terms of its throughput, latency, and client CPU requirements. Server CPU is not typically a limiting factor, since file server resources can be scaled independently of the network or file system clients. DATs may be built using a variety of underlying physical networks, including 1 and 2 Gbps Fibre Channel, 1 Gbps and 10 Gbps Ethernet, InfiniBand, and various proprietary interconnects. The newer 10Gbps interconnects (e.g. 4X InfiniBand and 10Gbps Ethernet) approach the limits of today's I/O busses. In addition, multiple interfaces may be trunked. As such, network interface throughput is not a performance limiting factor.

We break storage latencies down into three additive components: the delay in moving requests and responses between the client machine and the transport wire, the round trip network transit time, and the time required to process a request on the server. In the case of disk-based storage, the third component is a function of the disks themselves. Being mechanical, their access times dominate the overall access latency. Caching on the file server mitigates the latency problem by avoiding the disk component entirely for a fraction of client requests. Asynchrony or multi-threading can hide I/O request latency by allowing the application to do other work, including pipelining of additional requests while an I/O request is in progress.

The remaining metric is the client CPU overhead associated with processing I/O. If the CPU overhead per I/O operation is high, then for typical record sizes (2–16 KB), the client can spend a significant fraction of its available CPU time handling I/O requests. Additionally, client CPU saturation can impact throughput even for bulk data-movement workloads that can otherwise tolerate high latencies. The most significant performance advantages of DAFS stem from its attention to minimizing client CPU requirements for all workload types.

There are several major sources to CPU overhead, including operating system, protocol parsing, buffer copying, and context switching. Modern TCP/IP offload engines and intelligent storage adapters mitigate some of the protocol parsing and copying overhead, but do little to reduce system overhead. Without additional upper layer protocol support within the offload engines, incoming data will not be placed in its final destination in memory, requiring a data copy or page flip [7]. It is here that DAFS shines; only direct data placement avoids the costly data copy or page flip step in conventional network file system implementations. Previous results comparing DAFS direct data placement to offloaded NFS with page flipping show that the RDMA-based approach requires roughly 2/3 of the CPU cycles for 8KB streaming reads, improving to less than 1/10 of the cycles as the block size increases beyond 128KB [23, 24].

3 The DAFS Protocol

This section presents the Direct Access File System architecture and wire protocol. It begins with an overview of the DAFS design, followed by three subsections that cover performance improvements, file sharing semantics targeting the requirements of local file sharing environments, and security considerations.

The DAFS protocol uses a session-based client-server communication model. A DAFS session is created when a client establishes a connection to a server. During session establishment, a client authenticates itself to the server and negotiates various options that govern the session. These parameters include message byte-ordering and checksum rules, message flow control and transport buffer parameters, and credential management.

Once connected, messages between a client and server are exchanged within the context of a session. DAFS uses a simple request-response communication pattern. In DAPL, applications must pre-allocate transport buffers and assign them to a session in order to receive messages. Figure 1 shows a typical arrangement of these transport buffers, called *descriptors*, on both a client and file server. Messages are sent by filling in a *send descriptor* and posting it to the transport hardware. The receiving hardware will fill in a pre-allocated *receive descriptor* with the message contents, then dequeue the descriptor and inform the consumer that a message is available. Descriptors may contain multiple segments; the hardware will gather send segments into one message, and scatter an incoming message into multiple receive segments.

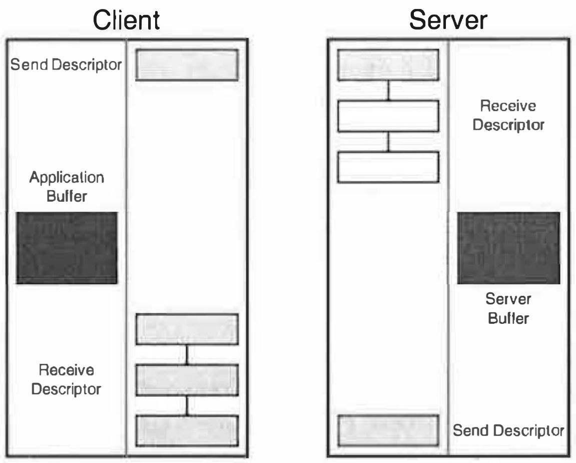


Figure 1: Client and server descriptor layout. The faint line in each system separates the provider library from its consumer, which contains an application data buffer. This and subsequent diagrams will use dark shading to show bulk data, light shading for protocol meta-data, and empty boxes for unused descriptor segments. In this diagram the receive descriptors are shown to have multiple segments; the transport hardware will scatter incoming messages into the segment list.

DAPL provides no flow control mechanism, so a DAFS server manages its own flow control credits on each open DAFS session. When a DAFS connection is established, the client and server negotiate an initial number of request credits; a client may only issue a request to a server if it has a free credit on an open session. Credit replenishment is not automatic; the server chooses when to allocate new credits to its clients.

DAFS semantics are based on the NFSv4 protocol. NFSv4 provides a broad set of basic file system operations, including file and directory management (*lookup*, *open*, *close*, *create*, *rename*, *remove*, *link*, *readdir*), file attribute and access control (*access*, *getfh*, *getattr*, *setattr*, *openattr*, *verify*, *secinfo*, *setclientid*), and data access (*read*, *write*, *commit*). All of these DAFS operations behave like their NFSv4 analogues. Most DAFS messages are small, so servers can allocate minimal transport resources waiting for incoming client requests. Instead of being included as part of the request or response, most bulk data is transferred using RDMA reads and writes. These RDMA operations are always initiated by the server, after a client sends a *handle* to a particular application buffer along with a request for the server to write data to (DAFS read) or read data from (DAFS write) that buffer. Always initiating the RDMA on the server minimizes the cross host memory management issues in a request-response protocol.

3.1 Performance Improvements

This subsection outlines key DAFS protocol features, with emphasis on how they improve file access performance and reduce client CPU utilization.

Message format DAFS simplifies the composition and parsing of messages by placing the fixed size portion of each message at the beginning of the packet buffer, aligned on a natural boundary, where it may be composed and parsed as fixed offset structures. Clients also choose the byte order of header information, moving the potential cost of byte swapping off of the client.

Session-based authentication Since DAFS communication is session-based, the client and server are authenticated when a session is created, rather than during each file operation. DAFS relies on the notion that a DAT connection is private to its two established endpoints; specific DAT implementation may guarantee session security using transport facilities. DAFS sessions also permit a single registration of user credentials that may be used with many subsequent operations. This reduces both the basic CPU cost of file operations on the server, and the latency of operations that require expensive authentication verification. Section 3.3 describes the DAFS authentication mechanisms in greater detail.

I/O operations For most bulk data operations, including *read, write, readdir, setattr,* and *getattr*, DAFS provides two types of data transfer operations: *inline* and *direct.* Inline operations provide a standard two-message model, where a client sends a request to the server, and then after the server processes the request, it sends a response back to the client. Direct operations use a three-message model, where the initial request from the client is followed by an RDMA transfer initiated by the server, followed by the response from the server. The transport layer implementation on the client node participates in the RDMA transfer operation without interrupting the client. In general, clients use inline data transfers for small requests and direct data transfers for large requests.

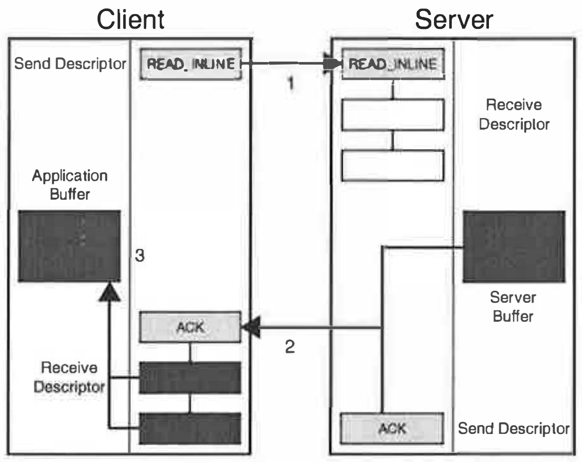


Figure 2: A DAFS inline read.

Figure 2 shows a typical inline read. In step 1, the client sends the `READ_INLINE` protocol message to the server. That message lands in a pre-allocated receive buffer, and in this case only requires a single segment of that buffer. In step 2, the server replies with a message containing both an acknowledgement and the bulk read data. The server uses a gathering post operation to avoid copying the bulk data. The client receives this message into its receive descriptor. Here, the message fills multiple segments. Finally, in step 3, the client copies the bulk data from the filled descriptor into the application's buffer, completing the read.

The DAFS protocol provides some support for replacing the copy in step 3 with a page flip by adding optional padding of inline read and write headers. For an inline read, the client requests the necessary padding length in its read request. In the case shown in Figure 2, the client would specify a padding such that the read inline reply plus the padding length exactly fills the first posted segment, leaving the bulk data aligned in the remaining segments.

Direct operations, on the other hand, offer the chief benefits of CPU-offload from the RDMA operation itself, and the inherent copy avoidance due to direct

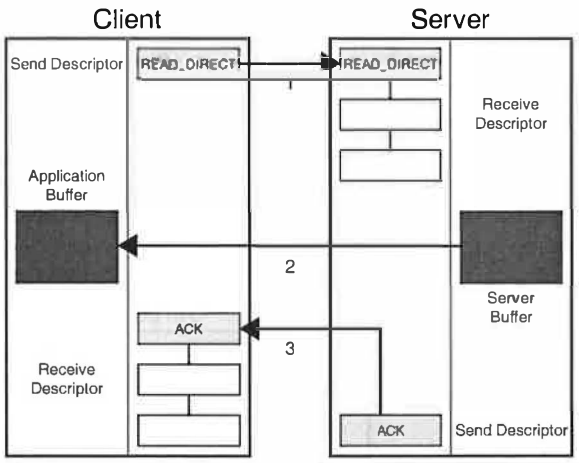


Figure 3: A DAFS direct read.

data placement (DDP) resulting from the separation of the request operation header (transferred in pre-allocated transport buffers) from request data (transported memory-to-memory via the RDMA transfer). The direct operations can also be used to transfer unusually large amounts of file meta-data (directory contents, file attributes, or symlink data). Figure 3 shows the three steps of a direct read. As with an inline read, the process begins with the client posting a short message, READ_DIRECT in this case, to the server. This message includes a list of memory descriptors that contain all the required information for the server to initiate an RDMA write directly into the client's application buffer. In step 2, the server gathers together the bulk data and issues an RDMA write operation to the transport. The client receives no information about this operation, so the transaction concludes with step 3, where the server sends a small acknowledgement message to the client. Since DATs preserve message ordering, the server can issue that acknowledgement immediately after the RDMA write; it need not wait for the RDMA to complete. Once the client receives the acknowledgement message, it is assured that the application buffer is filled in.

For latency-sensitive operations where transport costs dominate the request latency, the inline two-message mechanism may provide reduced overall latency, even though it requires a copy or page flip. For bulk data transfer, though, the direct data movement operations are preferred. Our user-space DAFS client always issues direct reads, since the client CPU costs are lower and the transport cost of an RDMA write is similar to a message post. Our client issues small writes as inline operations, though, since an RDMA read requires a transport-level acknowledgement and therefore has higher latency than an inline post.

By providing a small set of direct operations in addition to read and write, DAFS promotes the use of smaller inline buffers, thereby promoting additional operation concurrency. Effectively, DAFS is designed to use a large number of small pre-allocated buffers for most operations involving meta-data, and a variable number of variably sized RDMA buffers for bulk data transfer. For this reason, it is useful to determine the smallest useful buffer size. Since DAFS operations are generally issued in an unpredictable order, pre-allocated buffers are sized uniformly. In practice, since most meta-data operations require less than 1 KB of buffer space, the key trade-off in buffer size is determining whether read and write operations will be performed inline or direct. For a given memory resource cost, the number of concurrent requests can be increased dramatically by maintaining a small buffer size, and performing read and write requests using direct operations.

Reduced client overhead All RDMA transfers are initiated by the DAFS server. For file write operations, this means that the server has access to the write request header before the RDMA read transfer from the client begins. It can allocate an appropriately located, sized, and aligned buffer before initiating the transfer. This transfer should proceed as a zero-copy operation on both the client and server.

For file read operations, the server can initiate the RDMA write transfer to the client as soon as the data is available on the server. Since the address of the destination buffer on the client is contained in the read operation header, this operation should proceed as a zero-copy operation on both the client and server. Since DAPL preserves message ordering, the server may send the read response message without waiting for an RDMA completion. If the RDMA transfer fails for any reason, the connection will be broken and the response message will not arrive at the client.

DAFS is further designed to take advantage of the characteristics of Direct Access Transports to address the overheads from the operating system, protocol parsing, buffer copying, and context switching, leaving the client free to run application code. Some of these techniques are discussed in Section 4, which covers the user-space DAFS client API. The

DAFS protocol enables asynchronous data access, further lowering client overhead. Where the option exists, DAFS is somewhat biased towards reducing the client overhead even if it slightly increases file server overhead. One example of this tradeoff is permitting clients to use their native byte order.

Batch I/O facility DAFS provides a batch I/O operation, analogous to list I/O in some operating systems, that lets the client combine a number of read and write transfer requests, each accessing distinct buffers, files, and file offset ranges via RDMA transfers. The batch semantics match well with high-performance I/O APIs like MPI–IO [14]. Batch requests may specify memory scatter/gather regions on the client and file scatter/gather regions on the server. These regions need not align. Conceptually, the data on each side of the wire maps into a contiguous buffer for the transfer, though the transport may implement the scatter operation directly. Clients maintain control over the I/O pipeline and buffer resources by specifying the size of each batch of completion notifications. Clients may request synchronous or asynchronous notifications. In the former case, the server returns a single reply once the entire batch has completed. In the latter, the server asynchronously notifies the client as results complete; the client can advise the server how many completions should be grouped together before sending a completion notification for that group.

Batch I/O also allows the client to advise a maximum latency target for the batch, allowing a DAFS server to perform write-gathering and other local throughput optimizations while maintaining client control of the overall throughput. This technique is well suited for prefetching and asynchronous cleaning of dirty buffers in sophisticated client buffer managers like those in database engines. The latency of operations in this context is often secondary to efficiency of the batch, so applications that can tolerate high latencies can profit overall from batching requests and receiving asynchronous, batched results. If, after issuing a high-latency batch, a client finds it necessary to wait for a specific request to complete, it may issue an *expedite* operation to request that the server immediately process the specified request.

Cache hints DAFS client *cache hints* allow the file server to make better use of its own cache. Cache hints are advisory. Clients can specify file-level policy and access pattern information, similar to *madvise()*, as well as provide positive and negative byte-range hints with each read and write operation. In the local file sharing environment, network latencies and DAFS CPU requirements are low enough that server cache can be considered an extension of the client or application cache. Hints provide a mechanism in a highly tuned system for the client and server to better manage cache hierarchy. In cases like large database deployments where the client cache dwarfs the server cache, hints may allow the server to function as an effective metadata or victim cache instead of uselessly shadowing a fraction of the application's local cache.

Chaining DAFS *chaining* is similar to compound operations in NFSv4. NFSv4 defines compound requests to allow multiple file operations in a single network exchange as a way to reduce the combined latency of a series of operations. DAFS defines chained operations with the same semantics as NFSv4, but transfers the component operations separately. This approach limits the size of the preallocated buffers required to receive messages on both the client and server, and it preserves the benefits of a very simple packet layout for marshaling and parsing requests.

Combined with a user-space DAFS client, chaining permits client engines to implement complex operations that map to sequences of DAFS protocol requests. A DAFS client can then issue such requests without intervening between each protocol operation, reducing application impact while still preserving the failure characteristics of the equivalent series of operations.

3.2 New File Sharing Features

DAFS adds specific semantic enhancements to NFSv4 for high-performance, reliable clustered applications. These semantic features generally fall into two categories: shared access, and recovery semantics. Shared access semantics include a fencing mechanism to support application sharing within clusters and a shared key mechanism to arbitrate per-file access among cooperating applications. Recovery semantics include additional locking mechanisms and exactly-once failure semantics that aid recovery

Rich locking semantics File locking is important for cooperating applications accessing a common file store. These applications may crash while holding locks or otherwise fail to properly release their locks before terminating. Limited to existing locking APIs in POSIX and Windows, NFSv4 provides time-based lease expiration on locks, allowing other clients to access the resource without requiring client recovery. However, in these situations, the state of the previously locked file is suspect: presumably the lock was obtained in order to create atomicity across multiple operations, but since the lock was not explicitly released, the series of operations may be only partially completed, possibly leaving the file in an inconsistent state. Some recovery action may be necessary before making use of the data in the file. DAFS includes richer locking semantics that address this shortcoming and provide greater utility to applications.

DAFS persistent locks provide notification of an error condition following abnormal (lease expiration) release of a lock. Earlier *lockd*-based locks persisted following the failure of a client holding a lock, until either the client recovered or manual administrative intervention was performed. Further, due to UNIX file locking semantics, NFS clients in practice release locks immediately upon any termination of the owning process. Persistent locks provide notification of abnormal lock release to any subsequent client attempting to lock the file, until the lock is specifically reset. DAFS autorecovery locks provide an alternative safeguard against data corruption by associating data snapshot and rollback mechanisms with the act of setting a lock or recovering from an abnormally released lock.

Cluster fencing In clustered application environments, cluster membership is governed by a cluster manager that ejects nodes suspected of misbehaving or having crashed. Such a manager requires a mechanism whereby cluster members that are ejected from the cluster can be prevented from accessing shared file storage. The DAFS *fencing* operations manage client fencing access control lists associated with files or entire file systems that are under the control of the cluster application. The cluster manager populates the fencing list with cluster members' names, allowing multiple cluster applications to run on the same nodes independently of each other. Updates to a fencing list cause the DAFS server to complete any in-progress operations on the file(s) and update the appropriate access control list before responding to the fence request. Subsequent file operations from cluster members whose access privilege has been removed are denied by the DAFS server. The session-based architecture maps very conveniently to the fencing model.

The fence operation provides a serialization point for recovery procedures by the clustered application, without interfering with other files on the DAFS server or other client applications. The DAFS fencing mechanism is independent of standard file access controls, and is designed to support cooperating cluster members, similar to NFS advisory lock controls.

Shared key reservations NFSv4 allows share reservations, similar to those of CIFS, as part of the *open* operation. Together with an access mode of *read*, *write*, or *read-write*, a deny mode of *none*, *read*, *write*, or *read-write* may be specified to limit simultaneous access to a given file to those uses compatible with that of the process doing the open. DAFS enhances NFSv4 reservations by also providing *shared key reservations*. Any client opening a file can supply a shared key. Subsequent opens must provide the same key, or they will be excluded from accessing the file. This provides a similar capability to fencing at the file level. Key reservations have the scope of a file open. The duration of enforcement of a shared key reservation is from the time the first open specifies the key, to the time the file is finally closed by all clients that opened the file with that key.

Request throttling The use of credit-based message transfers managed on a per-client session basis allows the DAFS server to dedicate a fixed amount of resources to receiving client requests, and constantly redistribute them among many connected clients depending on their importance and activity level. Clients may affect this by requesting more credits as their workloads increase or returning unneeded credits to the server.

Exactly-once failure semantics To reach parity with local file systems, DAFS servers may implement exactly-once failure semantics that properly handle request retransmissions and the attendant problem of correctly dealing with retransmitted requests caused by lost responses. Retry timeouts do not offer this capability [17]. Our approach takes advantage of the DAFS request credit structure to bound the set of unsatisfied requests. The server's response cache stores request and response information from this set of most recently received requests for use during session recovery. Upon session reestablishment after a transport failure, the client obtains from the server the set of last-executed requests within the request credit window. It can therefore reliably determine which outstanding requests have been completed and which must be reissued. If the server's response cache is persistent, either on disk or in a protected RAM, then the recovery procedure also applies to a server failure. The result is to provide true exactly-once semantics to the clients.

Other enhancements The DAFS protocol includes a variety of additional data sharing features, including improved coordination for file append operations between multiple clients and the ability to automatically remove transient state on application failure. DAFS supports two common examples of the latter: creation of *unlinked files* that do not appear in the general file system name space, and *delete on last close* semantics, which provide for a deleted file that is currently being accessed to have its contents removed only when it is no longer being accessed.

3.3 Security and Authentication

The DAFS protocol relies on its transport layer for privacy and encryption, using protocols like IPSEC, and so contains no protocol provision for encryption. Since DAFS clients may be implemented in user space, a single host may contain many clients acting on behalf of different users. DAFS must therefore include a strong authentication mechanism in order to protect file attributes which depend on identifying the principal. For this reason, authentication is available in several extensible levels. For more trusted deployments, DAFS provides simple clear-text username/password verification, as well as the option to disable all verification. For environments that require stronger authentication semantics, DAFS uses the GSS-API [21] to support mechanisms such as Kerberos V [19].

4 The DAFS API

In addition to the DAFS protocol, the DAFS Collaborative defined the DAFS API, which provides a convenient programmatic interface to DAFS. The API provides access to the high-performance features and semantic capabilities of the DAFS protocol. The DAFS API is designed to hide many of the details of the protocol itself, such as session management, flow control, and byte order and wire formats. To the protocol, the DAFS API adds session and local resource management, signaling and flow control, along with basic file, directory, and I/O operations. The DAFS API is intended to be implemented in user space, making use of operating system functions only when necessary to support connection setup and tear down, event management, and memory registration. Through the use of kernel-bypass and RDMA mechanisms, the primary goal of the DAFS API is to provide low-latency, high-throughput performance with a significant reduction in CPU overhead. That said, a DAFS client may also be written as a kernel-based VFS, IFS, or device driver. Applications written to the POSIX APIs need not be modified to use those flavors of DAFS clients, but in general will lack access to advanced DAFS protocol capabilities.

The Collaborative did not attempt to preserve the traditional POSIX I/O API. It is difficult to precisely match the semantics of UNIX I/O APIs from user space, particularly related to signals, sharing open descriptors across *fork()*, and passing descriptors across sockets.[1] Other aspects of the POSIX API are also a poor match for high-performance computing [25]. For example, implicit file pointers are difficult to manage in a multi-threaded application, and there are no facilities for complicated scatter/gather operations. Unlike POSIX-based DAFS clients, the DAFS API allows the application to specify and control RDMA transfers.

The DAFS API is fully described in the DAFS API specification [8]; this section focuses on how the DAFS API provides inherent support for asynchronous I/O, access to advanced DAFS features like completion groups, registration of frequently-used I/O buffers, and improved locking and sharing semantics crucial for local file sharing applications. The DAFS API differs from POSIX in four signifi-

[1] One might say that 99% API compatibility doesn't mean 99% of applications will work correctly. It means applications will work 99% correctly.

cant areas:

Asynchrony The core data transfer operations in the API are all asynchronous. Applications written to use a polling completion model can entirely avoid the operating system. Asynchronous interfaces allow efficient implementations of user-space threading and allow applications to carefully schedule their I/O workload along with other tasks.

Memory registration User memory must be registered with a network interface before it can be made the destination of an RDMA operation. The DAFS API exports this registration to the application, so that commonly-used buffers can be registered once and then used many times. All of the data transfer operations in the DAFS API use *memory descriptors*, triplets of buffer pointer, length, and the memory handle that includes the specified memory region. The handle can always be set to NULL, indicating that the DAFS provider library should either temporarily register the memory or use pre-registered buffers to stage I/O to or from the network.

Completion groups The traditional UNIX API for waiting for one of many requests to complete is *select()*, which takes a list of file descriptors chosen just before the select is called. The DAFS API supports a different aggregation mechanism called *completion groups*, modeled on other event handling mechanisms such as VI's completion queues and Windows completion ports. Previous work has shown the benefits of this mechanism compared to the traditional *select()* model [2, 6]. If desired, a read or write request can be assigned to a completion group when the I/O request is issued. The implementation saves CPU cycles and NIC interrupts by waiting on predefined endpoints for groups of events instead of requiring the event handler to parse desired events from a larger stream.

Extended semantics The DAFS API provides an opportunity to standardize an interface to DAFS capabilities not present in other protocols. These include:

- Powerful batch I/O API to match the batch protocol facility. The batch I/O operation issues a set of I/O requests as a group. Each request includes a scatter/gather list for both memory and file regions. Batch responses are gathered using completion groups.
- Cancel and expedite functions. These are particularly useful for requests submitted with a large latency.
- A fencing API that allows cooperative clients to set fencing IDs and join fencing groups.
- Extended options at file open time, including cache hints and shared keys.
- An extensible authentication infrastructure, based on callbacks, that allows an application to implement any required security exchange with the file server, including GSS-API mechanisms.

5 Performance

This section presents experimental results achieved with our client and server implementation. The first results are micro-benchmarks demonstrating the throughput and client CPU benefits of DAFS. The second result is a demonstration that the DAFS API can be used to significantly improve an application's overall performance; in this case, *gzip*.

We have implemented the DAFS API in a user-space DAFS library (henceforth *uDAFS*). Our uDAFS provider is written to the VIPL API, and runs on several different VI implementations. We have also implemented a DAFS server as part of the Network Appliance ONTAP system software. The NetApp uDAFS client has been tested against both our DAFS server and the Harvard DAFS server [22, 24].

Our tests were all run on a Sun 280R client, connected to a Network Appliance F840 with 7 disks. We tested two GbE-based network configurations, one for DAFS and one for NFS. The DAFS network used an Emulex GN9000 VI/TCP NIC in both the client and file server; this card uses jumbo frames on Gigabit Ethernet as its underlying transport. The NFS network used a Sun Gigabit PCI adapter 2.0 card in the client and an Intel Gigabit Ethernet card in the server, speaking NFSv3 over UDP on an error-free network. The Sun adapter does not support jumbo frames. All connections on both networks are point-to-point.

5.1 Micro-benchmarking

Our first benchmark compares the cost of reading from a DAFS server to reads from an NFSv3 server. We show the results for both synchronous (blocking) I/O and asynchronous I/O. In each case we present two graphs. The first compares overall throughput for varying block sizes, while the second compares the CPU consumption on the client. We stress that these tests exercise the bulk data movement mechanisms of the protocols and their transports, not the higher protocol semantics.

In order to focus the measurements on a comparison of the protocols, these experiments perform reads of data that is cached on the server. Our uDAFS client implementation does not cache data and we configure the NFS client stack to do the same by mounting with the `forcedirectio` mount option. Such an arrangement is not contrived; database engines are typically run in this fashion as host memory is better allocated to the higher level database buffer cache instead of a kernel buffer cache that would merely replicate hot blocks, and potentially delay writing them to stable storage.

All of these results are gathered using a benchmarking tool that allows us to select either the POSIX or DAFS API for I/O, and to perform the I/O using either synchronous interfaces (*read()* and *dap_read()*) or asynchronous interfaces (*aioread()*, *aiowait()*, *dap_async_read()*, and *dap_io_wait()*). When using DAFS, the program first registers its buffers with the DAFS provider. All DAFS reads are done using DAFS direct reads, that is, using RDMA instead of inline bulk data. The tool includes built-in high-resolution timing hooks that measure realtime clock and process time.

Synchronous performance Our first test compares synchronous read operations for NFSv3 and DAFS. Figures 4 and 5 show the results. As expected, DAFS requires fewer CPU cycles per operation. Direct data placement keeps the line flat as block size increases, while the NFS stack must handle more per-packet data as the size increases. The latency of synchronous operations limits throughput at smaller block sizes, but once client overhead saturates the CPU, the DAFS client can move more data over the wire.

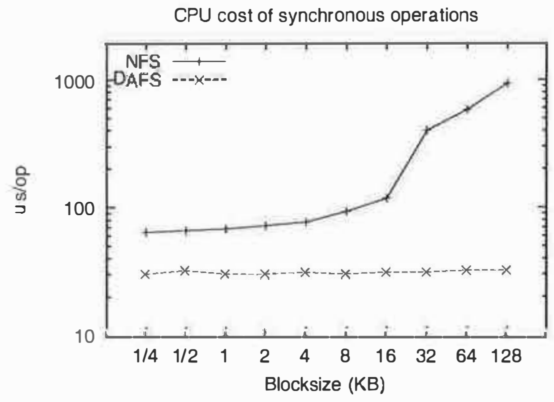


Figure 4: CPU time consumed per synchronous read request.

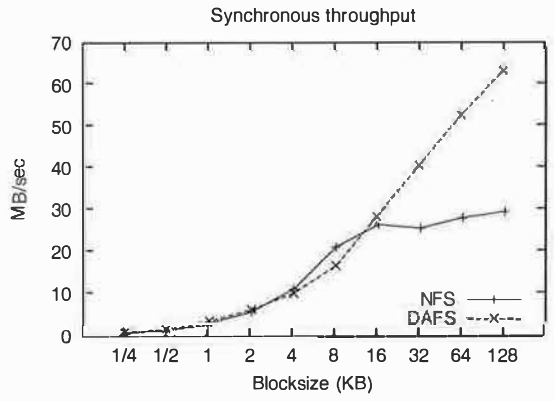


Figure 5: Synchronous throughput.

Asynchronous performance Figures 6 and 7 are a repeat of the previous tests, but use the asynchronous APIs to keep 32 reads in flight simultaneously. Here the advantages of DAFS become clearer. Asynchronous I/O improves throughput for both NFSv3 and DAFS, but the CPU time is the most significant result. While NFSv3 achieves high throughput only with an *increase* in CPU cost, DAFS requires *less* CPU time in asynchronous mode, since many results arrive before the client tries to block waiting for them. Asynchronous DAFS throughput approaches 85 MB/sec, matching our measured limit of 33 MHz PCI bus bandwidth on this platform.

5.2 GNU gzip

We converted the GNU *gzip* program to recognize DAFS filenames and use the DAFS API for access-

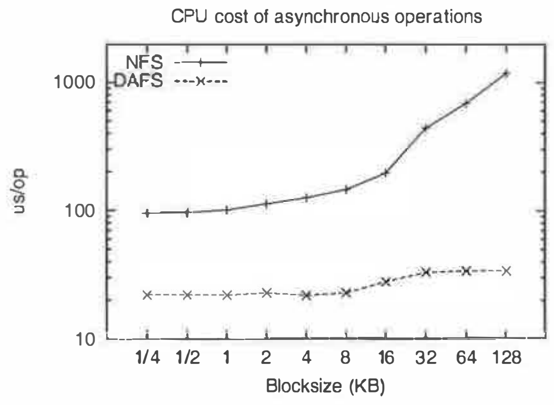


Figure 6: CPU time consumed per asynchronous read request.

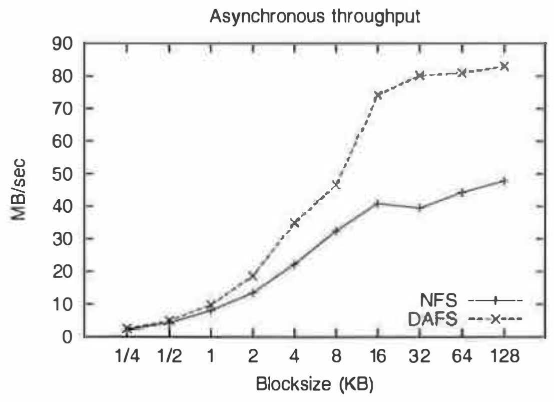


Figure 7: Asynchronous throughput.

ing those files. The conversion to the DAFS API adds memory registration, read-ahead, and write-behind capabilities. Figure 8 shows two comparisons. The test on the left measures the wall clock time of a single instance of *gzip* compressing a 550 MB file. Two factors account for the speedup. The *gzip* program uses 16 KB block sizes; Figures 4 and 6 show DAFS requiring 20 microseconds per operation at that block size, whereas NFSv3 consumes 100 microseconds. A 550 MB file corresponds to roughly 35,000 read operations, yielding nearly 3 seconds of CPU time saved just on the basis of read I/O cost. Moreover, during this test, the client CPU reported 0.4% idle time when running the DAFS version of *gzip* and 6.4% idle time when running the stock NFS version, accounting for a further 10 seconds of CPU time. By allowing asynchronous I/O without excessive CPU cost, the DAFS version hardly spends any time blocked waiting for data, so the CPU can spend more cycles generating the compressed blocks.

The second set of numbers compares the runtime of two *gzip* processes running in parallel, each operating on half of the full dataset. Here the uDAFS client demonstrates the advantages of operating system avoidance. The DAFS client achieves a nearly perfect 2X speedup, whereas the NFS versions are limited by kernel contention. In this case, both processors remained 100% busy while executing the DAFS version of *gzip*, but reported 34% idle time in the NFS case.

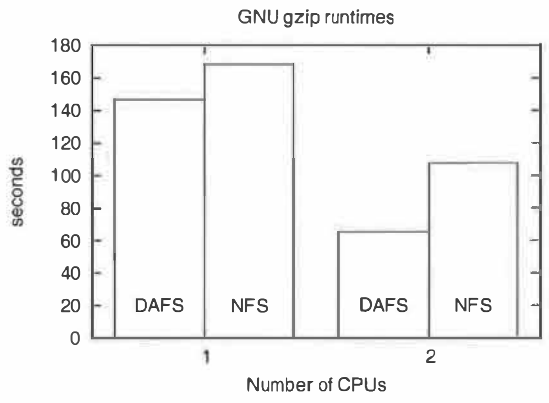


Figure 8: GNU gzip elapsed time.

6 Conclusions

The DAFS protocol enables high-performance local file sharing, and is targeted at serving files to clusters of clients. In this environment, the clients themselves are often application servers. DAFS takes advantage of Direct Access Transports to achieve high bandwidth and low latency while using very little client CPU. To support clustered environments, DAFS provides enhanced sharing semantics, with features such as fencing and shared key reservations, as well as enhanced locking. As an open client-server protocol, DAFS enables multiple interoperable implementations, while allowing for multiprotocol access to files on the server.

DAFS leverages the benefit of user-space I/O by providing asynchronous operations both in the protocol and in its application programming interface. DAFS shows significant measured performance gains over NFS on synchronous and asynchronous reads, and can yield a substantial performance improvement on I/O intensive applications, as demonstrated by our *gzip* experiment.

The unique combination of traits enabled by DAFS is extremely well-suited to the needs of local file sharing environments, such as data centers. The DAFS protocol is the basis for high-performance, scalable, and sharable network file systems that exploit current technology trends.

7 Acknowledgements

We'd like to acknowledge the contributions of many people from Network Appliance and all of the individuals representing the companies within the DAFS Collaborative in creating the DAFS protocol and API. We also wish to recognize the contributions of Somenath Bandyopadhyay, Tim Bosserman, Jeff Carter, Sam Fineberg, Craig Harmer, Norm Hutchinson, Jeff Kimmel, Phil Larson, Paul Massiglia, Dave Mitchell, Joe Pittman, Joe Richart, Guillermo Roa, Heidi Scott, and Srinivasan Viswanathan for their work on the file access protocol and API; Caitlin Bestler, Tracy Edmonds, Arkady Kanevsky, Rich Prohaska, and Dave Wells for their help in defining DAFS transport requirements; and Salimah Addetia, Richard Kisley, Kostas Magoutis, Itsuro Oda, and Tomoaki Sato for their early implementation experience and feedback.

We also thank Jeff Chase, our shepherd, and Mike Eisler for their valuable feedback.

References

[1] InfiniBand Trade Association. Infiniband architecture specification, release 1.0a, June 2001.

[2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[3] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[5] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*. USENIX Association, October 1996.

[6] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference*, pages 231–244, June 2001.

[7] Jeff Chase, Andrew Gallatin, and Ken Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[8] DAFS Collaborative. Direct Access File System: Application programming interface, October 2001.

[9] DAFS Collaborative. Direct Access File System protocol, version 1.0, September 2001.

[10] DAT Collaborative. uDAPL: User Direct Access Programming Library, version 1.0, July 2002.

[11] Compaq, Intel, and Microsoft. Virtual Interface Architecture specification, version 1.0, December 1997.

[12] Brian Pawlowski et al. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

[13] Internet Engineering Task Force. Remote direct data placement charter.

[14] MPI Forum. MPI-2: Extensions to the message-passing interface, July 1997.

[15] Robert W. Horst. Tnet: A reliable system area network. *IEEE Micro*, 15(1):37–45, 1995.

[16] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] Chet Juszczak. Improving the performance and correctness of an NFS server. In *USENIX Winter Technical Conference*, pages 53–63, January 1989.

[18] Mitsuhiro Kishimoto, Naoshi Ogawa, Takahiro Kurosawa, Keisuke Fukui, Nobuhiro Tachino, Andreas Savva, and Norio Shiratori. High performance communication system for UNIX cluster system. *IPSJ Journal Abstract*, 41(12 - 020).

[19] John Kohl and B. Clifford Neuman. Internet Engineering Task Force, RFC 1510: The Kerberos network authentication service (v5). September 1993.

[20] Ed Lee and Chandu Thekkath. Petal: Distributed virtual disks. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 84–93, October 1996.

[21] John Linn. Internet Engineering Task Force, RFC 2743: Generic security service application program interface, version 2, update 1, January 2000.

[22] Kostas Magoutis. Design and implementation of a direct access file system. In *Proceedings of BSDCon 2002 Conference*, 2002.

[23] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, and Margo Seltzer. Making the most of direct-access network-attached storage. In *Proceedings of the Second Conference on File and Storage Technologies (FAST)*. USENIX Association, March 2003.

[24] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo Seltzer, Jeff Chase, Richard Kisley, Andrew Gallatin, Rajiv Wickremisinghe, and Eran Gabber. Structure and performance of the Direct Access File System. In *USENIX Technical Conference*, pages 1–14, June 2002.

[25] Paul R. McJones and Garret F. Swart. Evolving the Unix system interface to support multithreaded programs. Technical Report 21, DEC Systems Research Center, September 1987.

[26] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[27] Sun Microsystems. Internet Engineering Task Force, RFC 1094: NFS: Network File System protocol specification, March 1989.

[28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[29] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*. USENIX Association, January 2002.

[30] SNIA. Common Internet File System (CIFS) technical reference, revision 1.0. *SNIA Technical Proposal*.

[31] Robert Snively. FCP draft, revision 8.0. *T11 Document X3T11/94-042v0*, 1994.

[32] Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Thomas Ruwart. The Global File System: A file system for shared disk storage. In *IEEE Transactions on Parallel and Distributed Systems*, October 1997.

[33] Raj Srinivasan. Internet Engineering Task Force, RFC 1831: RPC: Remote Procedure Call protocol specification version 2, August 1995.

[34] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP)*, pages 224–237, October 1997.

Making the Most out of Direct-Access Network Attached Storage

Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer
Division of Engineering and Applied Sciences, Harvard University

Abstract

The performance of high-speed network-attached storage applications is often limited by end-system overhead, caused primarily by memory copying and network protocol processing. In this paper, we examine alternative strategies for reducing overhead in such systems. We consider optimizations to remote procedure call (RPC)-based data transfer using either remote direct memory access (RDMA) or network interface support for pre-posting of application receive buffers. We demonstrate that both mechanisms enable file access throughput that saturates a 2Gb/s network link when performing large I/Os on relatively slow, commodity PCs. However, for multi-client workloads dominated by small I/Os, throughput is limited by the per-I/O overhead of processing RPCs in the server. For such workloads, we propose the use of a new network I/O mechanism, *Optimistic RDMA (ORDMA).* ORDMA is an alternative to RPC that aims to improve server throughput and response time for small I/Os. We measured performance improvements of up to 32% in server throughput and 36% in response time with use of ORDMA in our prototype.

1 Introduction

The performance of I/O-intensive applications using network-attached storage (NAS) systems over high-speed networks is often associated with high CPU and memory system overhead [3,6,9,12,20,23,29,30]. This overhead is primarily due to unnecessary memory copying and transport protocol processing, caused by inefficiencies in transporting file I/O traffic over general-purpose network protocol stacks. Memory copying is a per-byte source of overhead that limits the I/O bus throughput available for network transfers. Protocol processing, however, is primarily a per-I/O source of overhead. For example, in multi-client workloads dominated by small (4KB-64KB) I/Os, such as on-line transaction processing, remote memory paging [14], non-linear editing of video files, and standard office and engineering applications, performance can be limited by the server CPU, due to the per-I/O control transfer and processing overhead of RPC [34]. While overhead can be reduced with link-level and transport-level features offered by networks such as FibreChannel [18], this solution is not applicable to the widely deployed Ethernet and IP protocol infrastructure. In this paper, we explore alternative ways to reduce per-byte and per-I/O overhead in NAS systems over IP networks.

One approach to reduce per-byte overhead is to use network interface controller (NIC) support for transport protocol offload and for *remote direct data placement* (RDDP) [17]. An RDDP protocol performs network transfers directly to and from application buffers, eliminating the need for memory copying in the I/O data path. Remote direct memory access is a user-level networking [36] protocol achieving RDDP via remote memory read and write operations. The emergence of commercially-available NICs with RDMA capabilities has motivated the design of the *Direct-Access File System* (DAFS) [12,20], a network file access protocol optimized to use RDMA for memory copy avoidance and transport protocol offload. DAFS targets resource-intensive NAS applications, such as media streaming and databases.

In this paper, we argue that a simpler, alternative RDDP mechanism can offer similar memory copy avoidance and protocol offload benefits to those achieved with RDMA. This mechanism relies on *pre-posting of application buffers* at the receiver prior to the arrival of the RPC carrying the data payload [2]. This paper presents the first evaluation of a NAS system using this RDDP mechanism. Our results show that its benefits can be achieved with a kernel-based NFS client, whose two key properties are (a) support for optionally bypassing the kernel buffer cache, and (b) integration with the NIC for direct transfer to and from user-level buffers. A drawback of this approach, in contrast to the platform independent user-level client structure [20] enabled by DAFS, is that it is not as portable due to its dependence on specific kernel support.

While reduction of per-byte overhead is an important goal for NAS systems targeting I/O-intensive workloads, per-I/O overhead can limit performance of NAS servers involved in processing a large number of small I/Os issued by multiple clients. With the server CPU saturated due to the overhead of interrupts, scheduling, and file processing for small I/O RPCs, the NIC data transfer engine becomes underutilized, and as a result, throughput is less than the peak achievable by the network. In addition, server CPU involvement in each RPC increases file access response time. One way to improve throughput and response time for small I/Os is to *replace RPC by client-initiated RDMA*. Client-initiated RDMA does not involve the server CPU in setting up the data transfer, and therefore, has lower per-I/O overhead on the server compared to RPC.

This paper makes the following contributions:

(a) It shows that end-system overhead reduction for NAS applications is possible with simple RDDP support on NICs offering transport protocol offload.

(b) It differentiates between *throughput-intensive workloads performing large I/Os*, which primarily depend on RDDP for copy avoidance, and *workloads performing small I/Os*, for which client-initiated RDMA is necessary to reduce server per-I/O overhead.

(c) It proposes *Optimistic RDMA,* a new network I/O mechanism that enables client-initiated RDMA and benefits workloads performing small I/Os.

(d) It evaluates *Optimistic DAFS (ODAFS)*, our extension to DAFS that uses ORDMA, to improve server throughput and response time in workloads dominated by small I/Os.

The rest of this paper is organized as follows: In Section 2, we provide background and discuss related work. In Section 3, we present the implementations of the NAS systems that use RDDP. In Section 4, we describe the design and implementation of ORDMA and ODAFS. In Section 5, we use an experimental platform consisting of a Myrinet cluster of commodity PCs to evaluate the systems discussed in this paper.

2 Background and Related Work

Network storage systems can be categorized as Storage-Area Network (SAN)-based systems, which use a block access protocol, such as FibreChannel and iSCSI, or NAS-based systems, which use a file access protocol, such as NFS. SAN-based systems preserve an important property of direct-attached block I/O device interfaces, which is the ability for direct data transfers between the communication device and a user or kernel memory buffer. However, a drawback of using a SAN to share a storage volume is the need for additional synchronization mechanisms not present in current local file systems. Additionally, storage volumes accessed by user-level applications over a SAN are not under file system control and cannot be accessed using file system tools, complicating data management. In NAS-based systems, file servers handle sharing and synchronization. In addition, NAS storage volumes are under file system management and control.

High-performance NAS applications are becoming increasingly network I/O-intensive. This is due to the emergence of servers with large memory caches and the use of aggressive file caching and prefetching policies in conjunction with powerful disk I/O subsystems. In the future, new storage technologies reducing the $/MB ratio of stable storage, such as microelectromechanical systems, or MEMS, are expected to further ease the disk I/O bottleneck. On the other hand, network hardware performance is rapidly improving, with 2-2.5Gb/s commercial implementations available today and 10Gb/s implementations expected within a year. To deliver this network performance to applications, NICs should be able to transfer data at the speed of the network link. In addition, interaction with the host should take place with minimal CPU overhead. High-performance NICs are designed to integrate DMA engines able to transfer data between host memory and the network link at hardware speeds, for both large and small (4KB-64KB) I/Os [26]. Low CPU communication overhead is possible with user-level communication libraries [26,35,36] commonly used in distributed scientific computations. NAS systems, however, are usually implemented over general-purpose network protocols, such as Ethernet and TCP/IP, and communication abstractions, such as RPC, which result in high communication overhead.

A drawback of using RPC for file I/O data transfer is that this method requires staging of the data payload in intermediate host memory buffers and copying, to move the data to its final destination. One way to solve this problem is by enabling direct data transfers between clients and storage nodes over a SAN for large I/Os, as in several emerging clustered storage systems, such as Slice [3], MPFS–HighRoad [13], NASD [15,27], GPFS [30] and Storage Tank [16]. These systems use file servers for small I/O and metadata traffic. An alternative solution that does not require a SAN is to take advantage of RDDP mechanisms applicable to RPC-based data transfer over IP networks. For example, DAFS [12,20] and NFS-RDMA [9] are two recently proposed NAS systems based on NFS and using RDMA for memory copy avoidance and transport protocol offload. This approach promises to reduce communication overhead to levels comparable to that of block channel protocols.

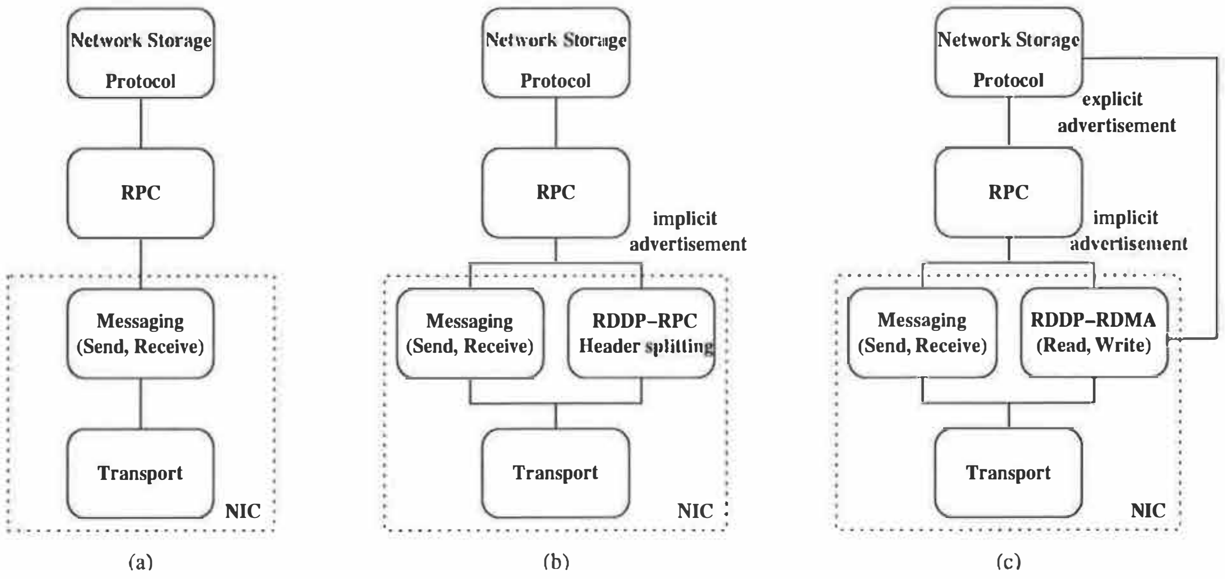


Figure 1. Protocol stack with the messaging and transport protocols offloaded to the NIC (a). RDDP is possible either by separating the data payload when in-lined in the RPC (b) or with RDMA (c).

In Section 2.1, we introduce network protocols that can be used to implement high-performance network-attached storage systems. In Section 2.2, we focus on the communication overhead of these protocols. Finally, in Section 2.3, we examine the impact of communication overhead on I/O throughput and response time.

2.1 Network storage communication protocols

Network storage systems can be implemented based on the interfaces and semantics of the network protocols shown in Figure 1. The primary communication abstraction is remote procedure call [5]. RPC can be implemented over a messaging layer, which can be offloaded to the NIC along with the transport protocol, as shown in Figure 1(a). The messaging layer can be accessed by the host via an interface that exports send and receive operations [7,35]. In addition, RDDP [17] enables direct placement of upper-level protocol data payloads into their target host memory buffers, as shown in Figure 1(b,c).

A communication layer implementing RDDP must perform the following operations: (1) Separate the protocol header from the data payload, (2) match the latter with its target buffer on the receiver, and (3) deposit it directly into its target buffer. To be able to perform (2), the target buffer must be *tagged* and *advertised* prior to the I/O. Tag advertisement can be either *implicit* or *explicit,* as shown in Figure 1, depending on whether it is performed by the RPC protocol or explicitly by the NAS protocol. In either case, however, advertisement is performed by an RPC. The data payload can be in-lined in the RPC message or transferred separately, using remote direct memory access.

RDDP using RPC (RDDP-RPC): One way to empower RPC with RDDP is to associate the target buffer with an RPC-specific tag and advertise this tag to the remote host. The remote host must include the advertised tag in the RPC that carries the data payload. The receiving NIC must match the tag with the target buffer, separate the data payload from the protocol headers (*header splitting*), and deposit the data directly into its target buffer. An RDDP-RPC mechanism evaluated in this paper is described in more detail in Section 2.2.

RDDP using RDMA (RDDP-RDMA): Another way to implement RDDP is using RDMA, which is a network data transfer protocol [8,37]. The RDMA layer exports a remote memory *read* and *write* interface. RDMA uses host virtual memory addresses as RDDP buffer tags. An RPC advertises the remote buffer and an RDMA moves the data to the target buffer. RDMA requires interaction with the upper-level protocol only to initiate the RDMA operation. It does not require interaction with the upper-level protocol at the target of the remote read or write operation. Only the RDMA initiator receives notification of completed events.

User-level networking [36] requires that RDMA use virtually addressed buffers. NICs with RDMA capabilities use a Translation and Protection Table (TPT), which is a device-specific page table, to translate virtual addresses carried on RDMA requests to physical addresses. To avoid limiting the size of the TPT, NICs can be designed to store the entire TPT in host memory, maintaining only a TLB on-board the NIC [26,37]. Systems using RDMA need to ensure that the NIC can find virtual to physical address translations of exported pages referenced in RDMA requests and that memory pages used for RDMA are kept resident in physical memory while the transfer takes place. Page *registration* through the OS is necessary in conventional NICs on the I/O bus, to ensure that address translations are available and that pages remain resident for the duration of the DMA.

Implications of RDDP tag advertisement. Protocols using RDDP for direct data placement typically advertise buffer tags by an RPC on a per-I/O basis. Advertisement of buffer tags on a per-I/O basis, however, means that both sides are involved in setting up each data transfer. An alternative that reduces the cost of per-I/O buffer advertisement is to cache advertisements in clients and carry file access operations by RDMA only [33]. Optimistic DAFS, our extension to DAFS described in Section 4.2, uses client-initiated RDMA without requiring buffer advertisement, thereby avoiding RPCs, on each I/O.

Messaging and Transport layers. The messaging layer exports a *queue pair* (QP) interface [7,35,36] for sending and receiving messages and for event notification. The messaging layer offers data transfer and event notification only, leaving event handling to upper-level protocols such as RPC. An example of a protocol providing user-level messaging and RDMA is the Virtual Interface (VI) architecture [35]. The transport layer exports a reliable, in-order stream abstraction similar to the TCP sockets interface. In addition, transport protocol support for framing, such as in SCTP [31], is required by RDDP in order to preserve upper-level protocol header and data payload boundaries.

2.2 Communication overhead

Host communication overhead in NAS end-system hosts is defined as the length of time that the host CPU is engaged in the transmission and reception of messages [10,11,22]. It consists of a *per-byte* component $o_{\text{per-byte}}$, which is the length of time that the CPU is engaged in data touching operations such as copying or integrity checking, and a *per-I/O* component $o_{\text{per-I/O}}$, which is the length of time that the CPU is engaged in processing the I/O request incurred in network and file system protocol stacks. The *per-packet* component, due to message fragmentation and reassembly, disappears if the transport protocol is offloaded to the NIC. We will assume an offloaded transport for the remainder of this paper. The following formula expresses the client or server CPU overhead of file access in an I/O transferring *m* bytes:

$$o(m) = m \times o_{\text{per-byte}} + o_{\text{per-I/O}}$$

There are a number of well-known techniques [10], such as checksum offloading, interrupt coalescing and increasing the network maximal transfer unit, for reducing overhead. These techniques are offered by several high-speed NICs and supported by mainstream operating systems. Further reductions in per-byte and per-I/O overhead are possible with the network I/O mechanisms and the NAS systems described in this paper and summarized in Table 1.

Reducing per-byte overhead. The primary source of per-byte overhead is memory copying. Avoiding unnecessary memory copying is a challenging problem since it requires either significant NIC support or significant file system and network protocol stack changes, such as integration of buffering systems [29,32] or virtual memory (VM) re-mapping techniques [6]. To avoid unnecessary copying, the I/O payload should be transferred directly from the source to the destination buffer. Avoiding memory copies on the outgoing path is relatively easy using scatter/gather support at the NIC or VM page re-mapping. Avoiding copies in the receiving path is more challenging since it requires NIC support to deposit incoming data either in a page-aligned location or directly at the final destination. In this paper we consider two ways to achieve direct data placement in host memory, either within the context of RPC or in combination with RDMA:

(a) RDDP-RPC. As described in Section 2.1, the RDDP-RPC protocol, which is NAS-specific, enables the NIC to identify and separate NAS and RPC headers from the data payload and deposit the latter directly into the target buffer on the host using DMA. In our implementation, we use the RPC transaction numbers as buffer tags. A tag is associated with an application buffer at the time when the latter is *pre-posted* by the receiving host, prior to sending the RPC request. Buffer tags are *implicitly advertised* in the context of the RPC protocol message exchange. RDDP-RPC imposes no buffer size or alignment restrictions on application buffers. Pre-posting of receive buffers (or *pre-posting*, for short) has previously been used in a kernel-resident RPC-based global shared memory service [2]. In Section 3.2, we describe a NAS system based on RDDP-RPC.

Network I/O mechanism	NAS system	Uses RDMA	Per-I/O tag advertisement
RDDP-RPC (§2.2)	NFS pre-posting (§3.2)	No	Yes
RDDP-RDMA (§2.2)	NFS hybrid (§3.1), DAFS [20]	Yes	Yes
Optimistic RDMA (§4)	Optimistic DAFS (§4.2)	Yes	No

Table 1. Network I/O mechanisms and NAS systems evaluated in this paper. RDDP mechanisms target per-byte overhead. Optimistic RDMA combines RDDP and per-I/O overhead reduction.

Untagged RDDP-RPC transfers are also possible and do not require pre-posting. The data payload is placed in intermediate, page-aligned host buffers and the physical memory pages of these buffers are re-mapped into the target buffer, provided that the latter is also page-aligned. A low overhead NFS implementation using header splitting and VM page re-mapping has been evaluated in a recent study [20].

(b) RDDP-RDMA. In this method, tag advertisement is performed using RPC but data transfer is performed using RDMA, as described in Section 2.1. RDMA imposes no buffer size or alignment restrictions. In Section 3.1, we describe NAS systems using RDDP-RDMA.

Both techniques rely on transport protocol offload to the NIC. They differ, however, in the complexity of implementation and in their generality. RDMA is a general-purpose data transfer mechanism: it is independent of any NAS protocol and exports a user-level API. NICs supporting RDDP-RPC are simpler to design and implement. They are customized, however, for particular NAS protocols and export a kernel API.

Reducing per-I/O overhead. The primary source of per-I/O CPU overhead is RPC processing. The main components of RPC are event notification, either by interrupt or polling, process scheduling, interaction with the NIC to start network operations or to register memory, and execution of the file protocol processing handlers. Part of the overhead of RPC is expected to improve with advances in core CPU technology. Other parts of the per-I/O overhead, however, such as interrupts and device control, are due to the interaction between the NIC and the host over the I/O bus and therefore not expected to improve as quickly as core CPU performance.

RDMA has fundamentally lower per-I/O overhead than RPC for remote memory transfers since it does not involve the target CPU. Reducing per-I/O overhead in file clients using RDMA is possible with techniques such as *batch I/O* in DAFS [12]. Using batch I/O, a single RPC is used to request a set of server-issued RDMA operations, amortizing the per-I/O cost of the RPC on the client. Reduction of per-I/O overhead on the file server is also important, perhaps even more so since servers receive I/O load from multiple clients. Our solution to reducing server per-I/O overhead uses client-initiated Optimistic RDMA, as discussed in Section 4.

2.3 I/O throughput and response time

Throughput and response time are standard I/O metrics used to assess performance in NAS systems. In this section we describe how CPU overhead affects these metrics.

Throughput is important for applications that can sustain several simultaneously outstanding transfers, either by having some knowledge of future accesses, or by involving a number of simultaneous synchronous activities, such as concurrent transactions in OLTP. From the overhead equation of Section 2.2 and with the per-byte component of overhead associated with memory copying eliminated using RDDP, overhead is dominated by its per-I/O component.

In addition to host CPU overhead, the performance of network storage applications may also depend on other parameters [11] such as the network link latency (L) and bandwidth (BW_{network}), and the NIC transfer rate (BW_{NIC}). Modern NIC architectures using DMA engines for transfers between the network link and host memory [26] ensure that the NIC is not the bandwidth bottleneck for messages larger than a certain threshold, i.e., $BW_{\text{NIC}} > BW_{\text{network}}$.

The I/O throughput achievable with a stream of I/O requests, each of size m, can be limited either by the network or by the (client or server) CPU:

$$\text{Throughput}\ (m) = \min\left\{BW_{\text{network}}, \frac{m}{o_{\text{per-I/O}}}\right\}$$

For large I/O blocks, even a low I/O request rate can saturate the network, and the throughput is determined by BW_{network}. For small I/O blocks, however, the CPU is more likely to become the resource limiting throughput. This is because the CPU is saturated processing RPCs at lower I/O rates than necessary to keep the NIC data transfer engine fully utilized. It is therefore important to reduce the per-I/O overhead for small file accesses. A previous study found that file server throughput in NFS workloads modeled by SPECsfs is most sensitive to host CPU overhead [23].

Besides throughput, response time is also important in transactional-style network storage applications that perform short transfers and cannot hide network latency using read-ahead prefetching or write-behind policies. Such applications usually have unpredictable access patterns involving small file blocks or file attributes. Response time is the delay to satisfy a remote file I/O request and consists of the transmission round-trip time on the network link, the NIC latencies, control and data transfer costs on the host I/O buses, and interrupt and scheduling costs in the case of remote procedure call-based I/O [34]. For a heavily loaded server, response time increases by the amount of queueing delays [23].

3 Direct transfer file I/O in NAS systems

File I/O in traditional operating systems is staged in the file system buffer cache, and memory copies are usually necessary to move data between network buffers, the file system cache and application buffers. In Section 2.2, we discussed network I/O mechanisms to achieve direct data placement and avoid the cost of data movement. In this section, we examine the use of those mechanisms to implement ***direct transfer file I/O***. This differs from what is commonly referred to as *direct file I/O* and associated with the `O_DIRECT` flag of the POSIX open system call. While direct file I/O implies a disabled file cache, which does not necessarily reduce memory copying, direct transfer file I/O additionally implies copy-free data transfer between the storage device and user-space buffers. This is easily achievable in local or network-attached storage systems, over parallel or serial SCSI, by programming the disk controller to DMA the requested data blocks directly to application buffers.

Direct transfer file I/O in network file systems is more challenging, as general-purpose NICs are not aware of upper-level transport protocol packet formats and semantics and cannot usually be programmed to DMA the data payload directly into application buffers. This is possible, however, with NIC support for RDDP-RDMA or RDDP-RPC.

To take advantage of a direct transfer I/O facility, file system clients must be modified so that their I/O operations bypass the buffer cache and propagate memory buffer information to the NIC. A drawback of using direct transfer file I/O is the need to register and pin user-level buffers, as shown in Figure 2. In the case of kernel file clients, registration has to happen on-the-fly and for each I/O to be transparent to user-level applications. One problem with this requirement is the possibility that the kernel may be unable, due to per-process resource limits, to pin the user-level buffers required for the transfer. Besides introducing additional failure modes, the need for on-the-fly memory registration and de-registration introduces a performance penalty in the data transfer path.

3.1 Direct transfer file I/O using RDDP-RDMA

One way to support direct transfer I/O is with RDDP-RDMA, used in the recently proposed DAFS [12] and NFS-RDMA [9] systems. DAFS is a file access protocol [20] that performs data transfers using server-initiated RDMA read and write operations, after explicitly advertising buffer addresses using RPC. In Sun's NFS-RDMA, buffer addresses are implicitly advertised by the RPC protocol. NFS-RDMA uses client- or server-initiated RDMA read operations issued from within the RPC protocol to pull data from remote buffers.

RDDP-RDMA requires registration and pinning memory buffers on both the client and the file server. This is a disadvantage not found in RDDP-RPC, which requires registration and deregistration only on the receiving side (e.g., the client in the case of reads). An advantage of RDDP-RDMA, however, is that the frequency of host interaction with the NIC can be reduced by caching registrations at the client and the server. With RDDP-RPC, NIC interaction is required on each I/O to pre-post application receive buffers.

In Section 5.1, we evaluate the performance of a kernel-based NFS-derivative system that performs data transfers using server-initiated RDMA. Our implementation modifies the NFS wire protocol to enable remote memory pointer exchange between client and server, like DAFS, but leaves the NFS client API unchanged, like NFS-RDMA. In Section 5.1, we refer to this system as *NFS hybrid*.

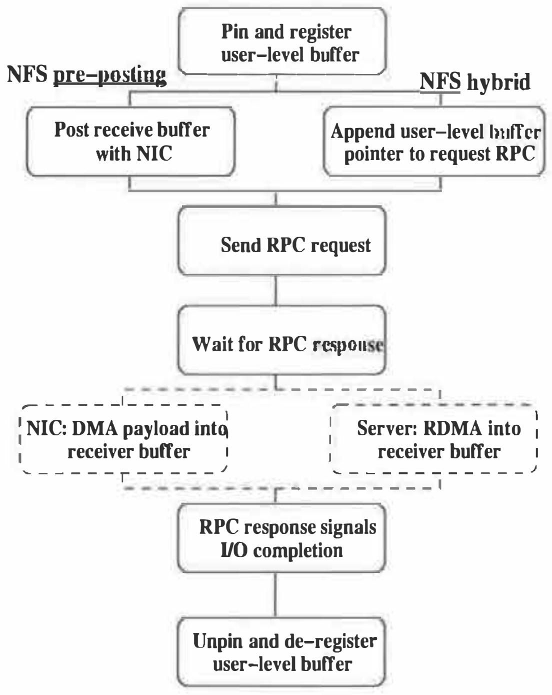

Pin and register user–level buffer
NFS pre–posting
NFS hybrid
Post receive buffer with NIC
Append user–level buffer pointer to request RPC
Send RPC request
Wait for RPC response
NIC: DMA payload into receiver buffer
Server: RDMA into receiver buffer
RPC response signals I/O completion
Unpin and de–register user–level buffer

Figure 2. NFS client actions for a read request with either RDDP-RDMA or RDDP-RPC.

3.2 Direct transfer file I/O using RDDP-RPC

Another way to support direct transfer I/O is with a NIC that supports RDDP-RPC. The implementation of an RDDP-RPC-based kernel client requires a device interface that communicates the following information to the NIC:

(a) A description of the user memory buffer, including the physical address pointing to the buffer, where data coming from the network is to be directly placed.

(b) A description of the request including the RPC transaction number and the type of request, enabling the NIC to recognize the data payload in the RPC response.

This scheme requires simple modifications in the *vnode* layer of existing network file clients to avoid the user/kernel copy, pin the user-level buffer in physical memory and give the NIC the description of the user-level buffer rather than a pointer to an intermediate buffer cache location. Both synchronous and asynchronous file I/O over an NFS client offering such support enjoys zero-copy, uncached data transfer.

One drawback of this scheme is that the NIC needs to be able to parse transport and application-level headers to understand RPC responses, which raises security and safety issues. These issues can be addressed by requiring supervisor privileges to program the NIC. Another drawback is that by bypassing the buffer cache, which abstracts the device layer, the file client is no longer part of the device-independent part of the kernel. Since not all NICs are expected to support an RDDP-RPC API, the file client depends on the availability of a device-specific API. However, making NIC-assisted direct transfer file I/O a mount option is expected to work well in practice.

This paper presents the first evaluation of a NAS system using RDDP-RPC. In Section 5.1, we refer to this system as *NFS pre-posting*.

4 Optimistic RDMA

The need for buffer tag advertisement on a per-I/O basis in RDDP systems requires the use of RPCs. These RPCs contribute to per-I/O CPU overhead, reducing server throughput and increasing response time in workloads dominated by small I/Os, as discussed in Section 2.3. One way to address these problems is to use *client-initiated RDMA,* without wrapping the RDMA in an RPC to prepare the server on a per-I/O basis. In this section, we introduce *Optimistic RDMA*, a novel network I/O mechanism that enables RDMA with these properties. The following design challenges must be addressed in an ORDMA mechanism:

Ensuring safety. One way to avoid accidental corruption or malicious buffer access by mutually untrusted clients is to use cryptographically strong hashing. Each exported memory segment is associated with a *capability* [24], which is a keyed message authentication code (MAC) computed and stored at the server TPT entry for the memory segment and given to the client. A capability protecting a memory segment is sent back to the server NIC with every ORDMA request for that segment. The server NIC verifies the validity of a capability before allowing a memory access. The server may revoke access privileges to an exported memory segment, for example, when protecting or invalidating VM page translations, by locally invalidating its capability in the TPT.

Handling remote memory access faults. Client-initiated RDMA may be faced with a number of exception conditions at the target NIC. For example, some of the targeted VM pages may no longer be resident in physical memory. In addition, targeted pages may be locked or protected. In the case of non-resident pages, one option is to enable the NIC to trigger a page-in disk I/O. However, this solution significantly increases the complexity of the NIC design and most importantly, it may not be supported by the OS. The ORDMA model enables clients to initiate RDMA that is guaranteed to succeed only if the target buffer is valid and exported by the server and is neither locked nor protected. In the opposite case, a recoverable access fault is signaled to the client by a network exception. After catching an ORDMA exception, a client handler may recover by retrying the access using an alternate access method, such as RPC.

Two important design choices in any ORDMA-based system are: (a) how a client finds references to server memory buffers, and (b) how a client handles exceptions due to failed ORDMAs. Section 4.2 describes the choices we have made in the Optimistic Direct Access File System.

4.1 ORDMA implementation

The two main ORDMA implementation issues are (a) how to synchronize between the NIC and the host CPU when accessing VM pages, and (b) how to report NIC–to–NIC network exceptions in case of remote memory access faults.

NIC–host CPU synchronization in accessing VM pages. Synchronization is necessary because the NIC is allowed to set up DMA transfers between the network and main memory, independently of the CPU. The kind of NIC–host CPU synchronization depends critically on OS support for multiple processors. An ORDMA-capable NIC in a multiprocessor OS can fully participate in the VM system, by pinning/unpinning and locking/unlocking VM pages in response to network events. This is because a multiprocessor OS offers the necessary synchronization structures for the NIC to appear indistinguishable from an additional CPU to the OS, except for its performance. On the other hand, a NIC in a uniprocessor OS may not be able to pin pages from interrupt handlers if, for example, the OS is non-preemptive. In this case, synchronization via the host memory resident TPT is necessary.

The NIC should ensure that the following two conditions hold for the duration of DMA: First, pages involved in DMA have to remain resident in physical memory. Second, conflicting accesses by another CPU or NIC should not be allowed. We chose to satisfy both requirements by treating VM pages with translations loaded in the NIC TLB as both pinned and locked. The alternative of locking pages only for the duration of an I/O requires frequent NIC–host CPU interaction and was deemed too expensive in the case of a NIC on the I/O bus. All pages in the TPT, except those with translations loaded on the NIC TLB, may be locked and invalidated by the host. The NIC updates the state of TPT entries by interrupting on each TLB miss. These interrupts increase CPU overhead but have the side-effect of speeding up the loading of TPT entries into the NIC, which is now done via a host-initiated programmed I/O operation, instead of (possibly several) NIC-initiated DMA on the PCI bus.

A drawback of having to synchronize via a device-specific page table is that the OS has to be aware of and adapt to the idiosyncrasies of the NIC. For example, it should always check with the NIC TPT before reclaiming a page and account for the fact that attempts to reclaim a physical page may fail until the page is evicted from the NIC TLB. To avoid starvation, the OS must increase its minimum free page threshold by the maximum amount of physical memory with page translations loaded on the NIC TLB. The OS must also be able to limit the effective size of the NIC TLB to avoid excessive pinning by the NIC.

NIC–to–NIC exceptions. ORDMAs may fail due to a variety of conditions, such as invalid address translation, protection violation, failure to lock page(s). We decided to support such exceptions by extending the VI protocol with recoverable RDMA failure semantics. Since VI is a layer on top of Myrinet's GM in our prototype, we first modified the Myrinet GM Control Program to report such conditions as exceptions in low-level *get* (i.e., RDMA read) and *put* (i.e., RDMA write) operations. These exceptions are reported as "soft" or recoverable transport errors in the VI descriptor status flags, and can be appropriately handled by higher-level software, such as the DAFS client and the ODAFS user-level cache described in Section 4.2.1.

4.2 Optimistic DAFS

The Optimistic Direct Access File System is our extension of the DAFS [12] protocol. Just like DAFS, ODAFS can use RPCs for all file requests. In addition to RPC requests, ODAFS clients may issue ORDMAs to directly access exported data and metadata buffers in the server file cache.

ODAFS is based on the following key principles:

(a) Clients maintain a *directory* or cache of remote references to server memory. These directories can be built either *eagerly* when clients ask the server for memory references, or *lazily* when the server piggybacks memory references with each RPC response.

(b) Directory entries need not be eagerly invalidated when the server invalidates VM mappings for exported references. Instead, invalid ORDMAs are caught at the server NIC, which throws exceptions reported to clients. An important advantage of this consistency mechanism is that the server does not need to keep track of clients caching memory references.

(c) The client is always prepared to catch an exception for each ORDMA operation. In such a case, the client issues an RPC to access the data.

Other important considerations for ODAFS clients are determining the size of the ORDMA directory, particularly in relation to the memory requirements for file data and attribute caching, and the replacement policies appropriate for maintaining the ORDMA directory. In this paper, we assume that the size of the ORDMA directory is small compared to the size of the data cache, and use the LRU replacement algorithm for ORDMA references. However, since ORDMA accesses are expected to be issued in response to client cache misses, a more appropriate strategy would be similar to the multi-queue algorithm for storage server caches [38].

4.2.1 ODAFS implementation

We implemented prototypes of an ODAFS client and server by extending the following existing DAFS components: a user-level DAFS file cache [1], a user-level DAFS API implementation [20] and a DAFS kernel server [21]. We rely on the ORDMA support for Myrinet described in Section 4.1.

The ODAFS server piggybacks remote memory references to data blocks in its kernel file cache onto RPC responses to the client. The ODAFS client stores these references in cache block headers. As data blocks are reclaimed by the client cache, memory references are allowed to live in "empty" headers. The client cache is configured with many more empty headers than data blocks. Ideally, it should have enough buffer headers to be able to map the entire server physical memory available for file caching.

We also modified the DAFS API to allow passing of ORDMA references, and the DAFS client implementation to include ORDMA operations in its event loop. On ORDMA exceptions, the DAFS client retries the operation using RPC in order to guarantee success. At RPC completion, the fresh piggybacked reference to the server buffers is passed to the ODAFS client.

The ODAFS server maps file blocks on a private 64-bit virtual address map. This is to ensure that there is always enough virtual address space to map large amounts of physical memory for long periods of time. Thus, we ensure that NIC TLB invalidations are due to the OS reclaiming a VM page due to memory pressure and never due to having to share a small virtual address space. This 64-bit address space is addressable only by the NIC and never by the CPU. It is therefore independent of whether the CPU has a 32- or 64-bit architecture.

Ideally, the replacement algorithm used in the server NIC TLB should be the same as the algorithm used in the client ORDMA directory.

4.2.2 Benefits and limitations

ODAFS is targeted for workloads performing small I/Os. ODAFS is most beneficial with significant memory-to-memory I/O traffic, such as that caused by small files and attribute accesses, and high server cache hit rates. The benefit comes mainly from the low server CPU overhead of the ORDMA mechanism. However, there are a number of workload characteristics that limit the applicability of ORDMA, and consequently the effectiveness of ODAFS. These are:

Few remote memory accesses, e.g., when client caching is effective in locally satisfying most file requests [25]. Note that this factor reduces the usefulness of any remote file access protocol.

Low ORDMA success rate, i.e., low server cache hit rates. If many ORDMAs result in failure, ODAFS performance is similar to that of DAFS as the cost of ORDMA exceptions and subsequent RPCs is masked by the high latency of server disk I/O.

Many file accesses that cannot be satisfied via ORDMA. This could be because the remote memory location of the target data may not be exportable. Examples are directory name lookups, which require significant processing on the server besides the actual data transfer.

Small read–write ratio. Writes require the update of associated file state, such as time of last modification and file block status on the server, besides the actual data transfer. Append-mode writes are harder as they further require allocating disk blocks on the server, checking resource limits, and potentially serializing over concurrent appending accesses.

Low NIC TLB hit rates. Satisfying TLB misses for a NIC on the I/O bus can be significantly more expensive than for a CPU TLB. In addition, network storage working sets can be very large and access patterns may not have enough locality to render NIC TLBs effective.

Finally, mixing ORDMA- and RPC-based file access has implications on the atomicity of file I/O. RPC-based file access guarantees that the entire I/O operation is atomic by locking the entire file for the duration of the I/O. However, ORDMA-based file access guarantees that at most one memory word is read or written atomically. By using both access methods, ODAFS effectively offers ORDMA's atomicity semantics. For UNIX file I/O semantics, client applications should explicitly lock files for the duration of I/O.

5 Experimental Results

Our experimental setup consists of a cluster of four PCs each with a 1GHz Pentium III processor, 2GB SDRAM and the ServerWorks LE chipset. The PCs are connected via a 2Gb/s switch over full-duplex ports. Each NIC has a 200MHz LANai9.2 network processor with 2MB of on-board SRAM in 64MHz/66-bit PCI slots. PCI bus throughput is measured at 450MB/s. All PCs run FreeBSD 4.6. The LANai drivers and firmware are based on GM-2.0 alpha1 release featuring support for remote direct memory access *get* and *put* primitives. The VI library is based on the Myricom VI-GM 1.0 release. This is a host-based user-level library mapping VI operations to GM operations and used by the user-level DAFS client [20]. A kernel port of the VI library supports the DAFS/ODAFS server [21]. Ethernet emulation is implemented in the standard LANai GM-2.0 firmware and drivers and supports UDP and IP checksum offloading and interrupt coalescing. The Ethernet packet MTU is 9KB. GM data transfers, however, are fragmented and reassembled by the LANai using a 4KB MTU. The GM driver and firmware are modified as described in Section 3.2 for RDDP-RPC and Section 4.1 for ORDMA (except for capabilities, which are not yet supported in our implementation). *NFS pre-posting* and *NFS hybrid* are implemented by modifying the FreeBSD 4.6 kernel, as shown in Figure 2. *NFS pre-posting* uses the RDDP-RPC device interface. *NFS hybrid* uses GM *put* to perform server-initiated RDMA

Protocol	Roundtrip (us)	Bandwidth (MB/s)
GM	23	244
VI	23 poll	244
	53 block	244
UDP/Ethernet	80	166

Table 2. Baseline Myrinet performance. One-byte roundtrip time.

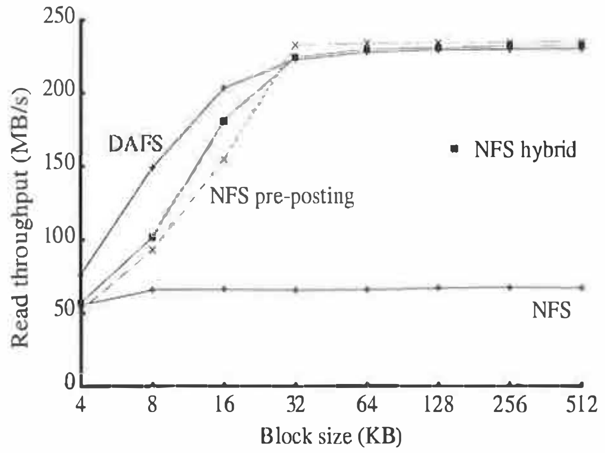


Figure 3. Client bandwidth performing read-ahead with variable application I/O block size.

writes to client memory buffers. Given the very low transmission error rates of Myrinet, we use UDP as our transport protocol to avoid the higher overhead of TCP. This configuration approximates the benefits of offloading TCP if it were supported by the NIC. Table 2 reports baseline network performance of the protocols used over the Myrinet network. These numbers are collected using the *gm_allsize*, *pingpong* and *netperf* programs for GM, VI-GM and UDP/IP protocols respectively.

5.1 Client overhead

In this section, we measure read throughput with a simple client and application performance with the Berkeley DB database.

Client read throughput. This experiment measures file read throughput with a simple client performing asynchronous read-ahead without any data processing. We compare DAFS to the two optimized NFS implementations, *NFS pre-posting* and *NFS hybrid*, and to standard NFS. The client reads data sequentially, using a varying block size, from a 1.5GB file warm in the server file cache. Read-ahead prefetching at the application level is done via the DAFS and POSIX *aio* APIs. NFS is mounted with the readahead parameter set to zero in all cases. UDP/IP is modified so that the NFS transfer size can match the application block size up to 512KB.

Figure 3 shows that for block sizes larger than 32KB DAFS can sustain read throughput of about 230 MB/s. As shown in Figure 4, it achieves this throughput consuming less than 15% of the client CPU for 64KB or larger blocks, by offloading the transport to the NIC and by being able to avoid all memory copies. Per-I/O overhead is progressively better amortized since the unit of data movement always matches the application block

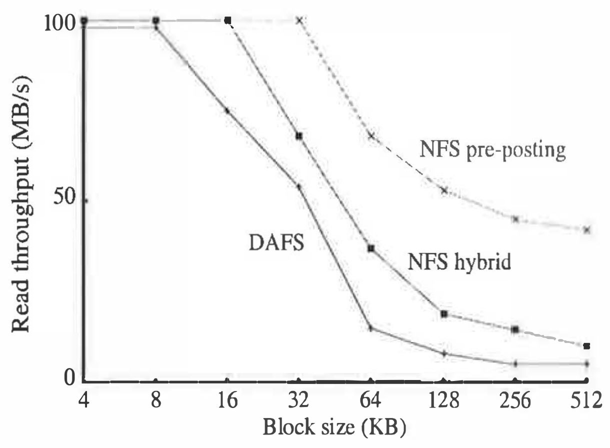


Figure 4. Client CPU utilization performing read-ahead with variable application I/O block size.

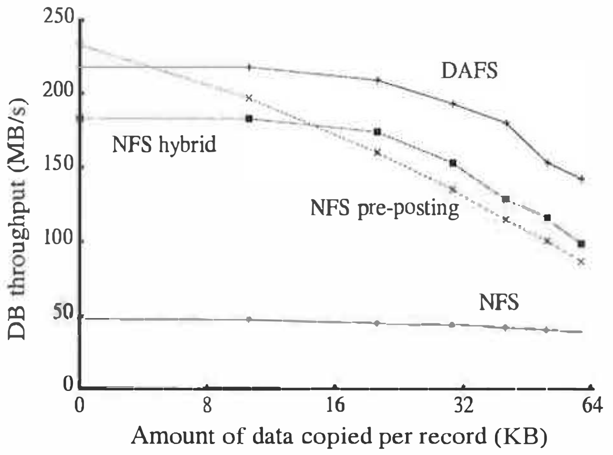


Figure 5. Berkeley DB performing asynchronous I/O.

size. For small block sizes, DAFS achieves low per-I/O overhead by using polling instead of interrupts. Similarly to DAFS, *NFS hybrid* sustains 230 MB/s for block sizes of 32KB or larger with CPU utilization dropping exponentially with increasing block size. However, even though both DAFS and *NFS hybrid* use RDMA, *NFS hybrid* uses more of the client CPU due to its higher per-RPC overhead. Both DAFS and the *NFS hybrid* clients avoid registering application buffers with the NIC on each I/O by caching registrations.

NFS pre-posting sustains 235 MB/s for block sizes 32KB or larger, performing data transfer in 8KB IP fragments. It slightly outperforms systems using RDMA because the size of Ethernet packets (8KB) is twice the size of the 4KB GM fragments. The decline in its client CPU utilization is eventually limited for large block sizes as the total number of IP fragments is independent of the block size. In addition, the *NFS pre-posting* client interacts with the NIC for pre-posting application receive buffers on each I/O. Standard NFS (not shown in Figure 4) achieves a maximum throughput of 65 MB/s, limited primarily by memory copying, which saturates the client CPU.

Berkeley DB performing asynchronous I/O. In this experiment, we use Berkeley DB to show the effect of client CPU overhead in application performance. Berkeley DB [28] (*db*) is an embedded database management system that provides recoverable, transaction-protected access to databases of key/data pairs. It is linked into the application address space and maintains its own user-level cache of recently accessed database pages. *Db* is modified to asynchronously prefetch database pages when it is possible to pre-compute a set of required pages.

In this experiment, an application uses *db* to compute a simple equality join with 60KB records. The result of the join is a large list of keys, retrieved from the database file located on the server. *Db* pre-computes the list of required pages and performs read-ahead, maintaining a window of outstanding I/Os. To vary the computational requirements of the application, we increase the amount of data copied from the *db* cache into the application buffer for each record, from one byte to 60KB, and report the application throughput in Figure 5. The throughput sustained by the application when there is little memory copying is close to the wire throughput for all systems except standard NFS. *NFS pre-posting* performs slightly better than the other systems, as is also the case in Figure 3. As the amount of copying increases, performance becomes limited by the client CPU. Relative system performance is inversely proportional to each system's client CPU overhead for 64 KB network I/O transfers.

5.2 Server I/O throughput and response time

In this section we present microbenchmark and PostMark results highlighting the properties of ORDMA and the upper bounds for performance improvements in ODAFS applications. In all cases, a file cache based on DAFS open delegations [12] is interposed between the application and the DAFS/ODAFS API. To avoid introducing platform-specific parameters, such as the cost of NIC memory registration and TLB misses, we ensure that RDMA is done on pre-registered buffers and always hits in the NIC TLB. The cost of a NIC TLB miss is about 9μs for ORDMA in our prototype. This penalty can be reduced in NICs that have large TLBs, are integrated on the memory bus, or share a TLB with the host CPU [4].

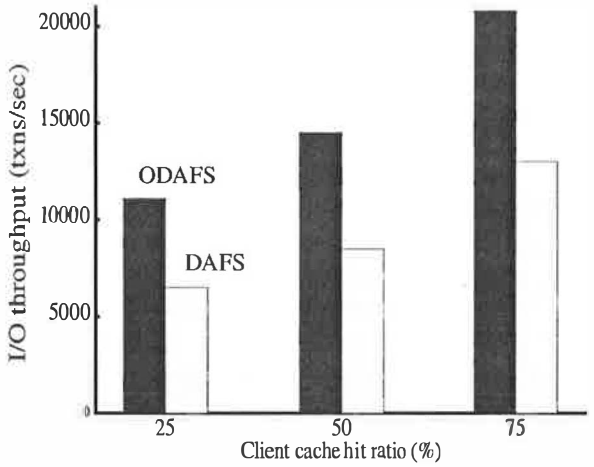


Figure 6. PostMark I/O throughput. Single client with variable cache hit ratio.

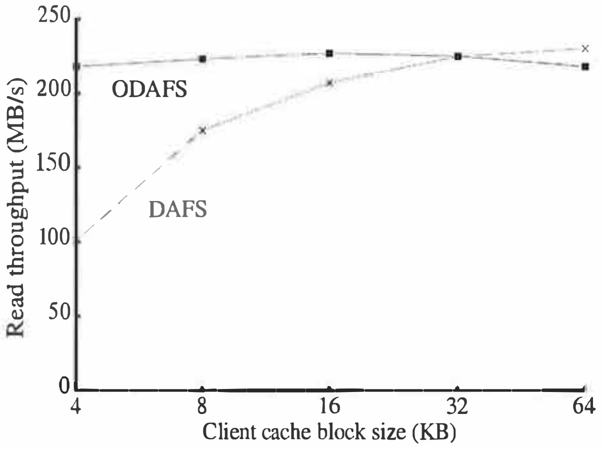


Figure 7. Server throughput. Two clients reading a large file using a large block size.

Microbenchmarks. We measure I/O response time in reading a 4KB block from server memory using (a) in-line RPC read, that is, the data payload in-lined with the RPC response, (b) direct RPC read, that is, the data payload transferred by server-initiated RDMA write, and (c) client-initiated ORDMA read. The file cache is configured with a small number of data blocks but with a large number of headers that can retain remote memory references. In this microbenchmark, a simple application sequentially reads a 1GB file warm in the server cache twice, in increments of 4KB. The client cache is configured with a 4KB block size and is cold prior to starting the experiment.

I/O mechanism	Response Time (us)	
	in mem.	in cache
RPC in-line read	128	153
RPC direct read	144	144
ORDMA read	92	92

Table 3. I/O response time with 4KB block size.

During the first pass, all I/O requests miss in the client cache, which, in response, initiates remote file accesses using either in-line or direct RPC. RPC responses carry remote memory references to file blocks on the server cache. During the second pass, I/Os still miss in the client cache. However, this time remote I/O may also be performed by ORDMA since the client cache managed to map the entire file on the server after having accessed it once during the first pass. Table 3 shows the I/O response time during the second pass using different network I/O mechanisms. RPC in-line involves a memory copy in the client from the communication buffers to the file cache. ORDMA yields about 36% lower response time than direct RPC.

Effect of client caching. In this experiment, we model a file client accessing a set of small files synchronously over DAFS and ODAFS. The file set size exceeds the client cache size in all cases. We model such a latency-sensitive workload by configuring the PostMark [19] benchmark for read-only transactions without file creations or deletions. Each read I/O is preceded by a file open and followed by a file close operation. After the first open of a file, which grants the client an open delegation, each subsequent open or close for that file is satisfied locally. We use a 4KB average file size and configure the client cache with a 4KB block size. The client cache hit ratio determines the frequency of remote memory access. By varying the size of the client cache and keeping the file set size constant we progressively increase its hit ratio from 25% to 50% to 75%. We find that in all cases ODAFS yields about 34% higher throughput than DAFS (Figure 6), reflecting the difference in response time between ORDMA and direct RPC. This is because, despite the benefit of client caching, overall performance is sensitive to the cost of remote memory accesses. The DAFS server CPU utilization drops from 30% to 25% to 20% as the client cache hit ratio improves. However, ODAFS uses no server CPU after it manages to collect remote memory references for the entire server cache, which occurs after the client has accessed each file at least once.

Server throughput. In this experiment, we show the effect of per-I/O overhead on server throughput. We model a multi-client, throughput-intensive workload dominated by small I/Os by configuring two clients to sequentially read a 1GB file warm in the server cache twice, using a large block size. For reads larger than the cache block size, the cache starts internal read-ahead up to the size of the application request. To vary the unit of network I/O, we progressively increase the cache block size from 4KB to 64KB and measure server throughput for each cache block size during the second pass, as shown in Figure 7. We find that with ODAFS, the two clients are able to saturate the server network link for all cache block sizes (except for 64KB due to a performance bug in GM *get*) without using the server CPU. DAFS yields lower server throughput for small I/O blocks, saturating the server CPU due to processing direct RPCs. For the smallest cache block size of 4KB for which the difference between DAFS and ODAFS is maximal, the DAFS server is primarily constrained by network interrupts. Switching to polling for all network events, DAFS throughput improves to about 170 MB/s reducing the performance improvement attainable from ODAFS to 32%.

6 Conclusions

In this paper, we show that two network I/O mechanisms for RDDP, *pre-posting application receive buffers* and *RDMA*, are effective in reducing per-byte CPU overhead in NAS end-systems. Our experiments show that they both enable a throughput-intensive streaming client to achieve file access at the speed of a 2Gb/s network link. RDMA offers the advantage of a general-purpose user-level API, enabling portable user-level implementations. Workloads dominated by small I/Os are more sensitive to per-I/O overhead. For such workloads, we propose a new network I/O mechanism, *Optimistic RDMA*, that aims to improve server throughput and response time. We have implemented a prototype of ORDMA and of *Optimistic DAFS*, our extension of the DAFS protocol that uses ORDMA. We measured improvement in server throughput and response time by up to 32% and 36%, respectively, in small I/O transfers.

7 Acknowledgements

This research was supported by Network Appliance and Sun Microsystems. We thank our shepherd Peter Corbett, Dan Ellard, Omri Traub, Matt Welsh, and the anonymous reviewers for their helpful comments. We also thank Andrew Gallatin of Myricom Inc. for his help with Myrinet software.

8 Software Availability

All NFS, DAFS and ODAFS software, including FreeBSD patches and Myrinet driver, library and firmware modifications, used in this paper is freely available from http://www.eecs.harvard.edu/dafs.

References

[1]. S. Addetia, “User-level Client-side Caching for DAFS”, Harvard University TR-14-01, March 2002.

[2]. D. Anderson, J. Chase, S. Gadde, A. Gallatin, K. Yocum, “Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet”, in *Proc. of USENIX Annual Technical Conference*, pp. 143-154, New Orleans, LA, June 1998.

[3]. D. Anderson, J. Chase, A. Vahdat, "Interposed Request Routing for Scalable Network Storage", in *Proc. of 4th USENIX OSDI Symposium*, pp. 259-272, San Diego, CA, October 2000.

[4]. B. Ang, D. Chiu, L. Rudolph, Arvind, “Message Passing Support on StarT-Voyager”, *CSG Memo 387*, MIT Laboratory for Computer Science, July 1996.

[5]. A. Birrell, B. Nelson, “Implementing Remote Procedure Calls”, *ACM Transactions on Computer Systems*, (2)1:29-59, Feb. 1984.

[6]. J. Brustoloni, “Interoperation of Copy Avoidance in Network and File I/O”, in *Proc. of the IEEE INFOCOM’99 Conference,* pp. 534-542, New York, NY, March 1999.

[7]. P. Buonadonna, D. Culler, “Queue-Pair IP: A Hybrid Architecture for System Area Networks", in *Proc. of 29th ISCA Symposium*, Anchorage, AK, May 2002.

[8]. G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, J. Wilkes, “An Implementation of the Hamlyn Sender-Managed Interface Architecture”, in *Proc. of 2nd USENIX OSDI Symposium*, pp. 245-259, Seattle, WA, October 1996.

[9]. B. Callaghan, NFS over RDMA, *Work in progress presented at 1st USENIX FAST Conference,* Monterey, CA, January 2002.

[10]. J. Chase, A. Gallatin, K. Yocum, “End System Optimizations for High-Speed TCP”, *IEEE Communications*, (39)4:68-74, April 2001.

[11]. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eicken, “LogP: Towards a Realistic Model of Parallel Computation”, in *Principles and Practice of Parallel Programming*, pp. 1-12, 1993.

[12]. M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, M. Whittle, “The Direct Access File

System", to appear in *Proc. of 2nd USENIX FAST Conference*, San Francisco, CA, March 2003.

[13]. EMC Cellera HighRoad, White Paper http://www.emc.com/pdf/proucts/celerra_file_server/HighRoad_wp.pdf, January 2002.

[14]. E. Felten, J. Zahorjan. "Issues in the Implementation of a Remote Memory Paging System", CS TR 91-03-09, University of Washington, March 1991.

[15]. G. Gibson, D. Nagle, K. Amiri, J. Buttler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, J. Zelenka, "A Cost-Effective, High-Bandwidth Storage Architecture", in *Proc. of 8th ASPLOS Conference*, pp. 92-103, San Jose, CA, October 1998.

[16]. D. Pease, IBM Storage Tank, *Work in progress presented at 1st USENIX FAST Conference*, Monterey, CA, January 2002.

[17]. IETF *Remote Direct Data Placement* (RDDP) Working Group, http://www.ietf.org/

[18]. C. Jurgens, "FibreChannel: A Connection to the Future", *IEEE Computer*, 28(8):88-90, August 1995.

[19]. J. Katcher, "PostMark: A New File System Benchmark", Network Appliance TR-3022, October 1997.

[20]. K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, E. Gabber, "Structure and Performance of the Direct Access File System", in *Proc. of USENIX Technical Conference*, Monterey, CA, pp. 1-14, June 2002.

[21]. K. Magoutis, "Design and Implementation of a Direct Access File System Kernel Server for FreeBSD", in *Proc. of USENIX BSDCon 2002 Conference*, San Franscisco, CA, pp. 65-76, February 2002.

[22]. R. Martin, A. Vahdat, D. Culler, T. Anderson, "Effects of Communication Latency, Overhead and Bandwidth in a Cluster Architecture", *Proc. of the 24th Annual ISCA*, pp. 85-97, Denver, Colorado, June 1997.

[23]. R. Martin and D. Culler, "NFS Sensitivity to High-Performance Networks", in *Proc. of SIGMETRICS '99/ PERFORMANCE '99 Joint International Conf. on Measurement and Modeling of Computer Sys.*, pp. 71-82, Atlanta, GA, May 1999.

[24]. S. Mullender and A. Tanenbaum, "The Design of a Capability-based Distributed Operating System", *The Computer Journal*, 29(4):289-299, 1986.

[25]. D. Muntz and P. Honeyman, "Multi-level Caching in Distributed File Systems (your cache ain't nuthin' but trash), In *Proc. of USENIX Technical Conference*, pp. 305-314, San Antonio, TX, January 1992.

[26]. Myricom LANai9.2 and GM communication library, Myricom Inc., http://www.myri.com

[27]. D. Nagle, G. Ganger, J. Butler, G. Goodson, C. Sabol, "Network Support for Network-Attached Storage", in *Proc. of Hot Interconnects*, Stanford, CA, August 1999.

[28]. M.Olson, K. Bostic, M. Seltzer, "Berkeley DB", in *Proc. of USENIX Technical Conference (FREENIX Track)*, pp. 183-192, Monterey, CA, June 1999.

[29]. V. Pai, P. Druschel, W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System", in *Proc. of 3rd USENIX OSDI Symposium*, pp. 15-28, New Orleans, LA, February 1999.

[30]. F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", in *Proc. of 1st USENIX FAST Conference*, Monterey, CA, January 2002.

[31]. R. Steward, C. Metz, "SCTP: New Transport Protocol for TCP/IP", IEEE Internet Computing, pp. 64-69, November 2001.

[32]. M. Thadani, Y. Khalidi, "An Efficient Zero-copy I/O Framework for UNIX", SMLI TR95-39, Sun Microsystems Lab, Inc., May 1995.

[33]. C. Thekkath, H. Levy and E. Lazowska, "Separating Data and Control Transfer in Distributed Operating Systems", *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, San Jose, CA, October 1994.

[34]. C. Thekkath and H. Levy, "Limits to Low-Latency Communication on High-Speed Networks", *ACM Trans. on Computer Systems*, 11(2):179-203, 1993.

[35]. Virtual Interface Architecture Specification, Version 1.0, http://www.viarch.org, December 1997

[36]. T. von Eicken, A. Basu, V. Buch and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", *Fifteenth ACM Symposium on Operating Systems Principles*, pp. 40-53, Copper Mountain Resort, CO, December 1995.

[37]. M. Welsh, A. Basu and T. von Eicken, "Incorporating Memory Management into User-Level Network Interfaces", in *Proc. of Hot Interconnects,* pp. 27-36, August 1997.

[38]. Y. Zhou, J. Philbin, K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches", in *Proc. of USENIX Technical Conference*, pp. 91-104, Boston, MA, June 2001.

Passive NFS Tracing of Email and Research Workloads

Daniel Ellard, Jonathan Ledlie, Pia Malkani, Margo Seltzer
{ellard,jonathan,malkani,margo}@eecs.harvard.edu
Division of Engineering and Applied Sciences, Harvard University

Abstract

We present an analysis of a pair of NFS traces of contemporary email and research workloads. We show that although the research workload resembles previously-studied workloads, the email workload is quite different. We also perform several new analyses that demonstrate the periodic nature of file system activity, the effect of out-of-order NFS calls, and the strong relationship between the name of a file and its size, lifetime, and access pattern.

1 Introduction

Trace-based analyses have guided and motivated contemporary file system design for the past two decades. The original analysis of the 4.2BSD file system [9] motivated many of the design decisions of the log-structured file system (LFS) [13]. The revisitation of the original BSD study [1] confirmed the community's earlier results and further drove file system design and evaluation towards the support of distributed file systems as well as computer science engineering and research workloads. In the late 1990's, our repertoire of trace-based studies was expanded to include the increasingly dominant desktop systems of Microsoft [12, 15] and new workloads such as WWW servers [12]. It is clear from the literature of trace-based studies that there are many interesting and important workloads to consider when designing a file system, and that new workloads emerge as new applications and uses for file systems appear. We believe that as the community of computer users has expanded and evolved there has been a fundamental change in the workloads seen by file servers, and that the research community must find ways to observe and measure these new workloads.

In this paper, we present an analysis of two contemporary NFS workloads: EECS, a CS research workload, and CAMPUS, the central computing facility for our university. Both of these systems exhibit new characteristics that future file system designs might be able to exploit.

The EECS workload extends the pattern of research workloads of earlier studies, but ventures into new territory – the EECS workload is dominated by metadata requests and has a read/write ratio of less than 1.0.

The CAMPUS workload is almost entirely email, and is dominated by reading. Nearly all of the active files on CAMPUS fall into one of four categories – mailbox files, lock files, scratch files used by mail clients, and configuration files. Each category of file has a predictable size, lifespan, and access pattern. Virtually all files on CAMPUS can be correctly categorized by their filename, and therefore filenames might provide useful hints to the file system about each file.

The contributions of this work are:

- A large set of anonymized traces from both technical/research and email workloads
- An analysis of these new traces and comparison to prior traces
- New techniques for analyzing NFS traces, including a new method for quantifying workload sequentiality and understanding its impact on file servers
- Evidence that many properties of a file are predicted by its name
- Tools to gather new anonymized NFS traces

The rest of this paper is organized as follows. Section 2 describes our tracing software and the trace anonymization process. In Section 3, we give an overview of the systems we studied and the traces we gathered. We discuss the advantages and challenges of gathering file system traces from NFS environments in Section 4. In Section 5, we describe our new traces and compare them to the results of previous trace-based studies, highlighting the similarities and differences. In Section 6, we discuss the new analyses and findings revealed by our traces, and in Section 7, we summarize our findings and discuss directions for future research.

2 Our Tracing Software

Our traces were collected by attaching a computer running our NFS packet snooping software (based on an extensively modified version of `tcpdump`) to a mirror port on the network switch hosting the servers under study. Our tracing software can handle any combination of NFSv2 and NFSv3, TCP or UDP transport, gigabit

Ethernet, and jumbo frames. Unlike other tracing tools, we also support some forms of TCP packet coalescing and automatic anonymization of the traces. Our software is also open-source, easy to configure, runs unattended, and is portable across different flavors of UNIX.

The output of our tracing software is either a time-stamped literal record for each NFS call and response observed over the network or an anonymized version of these records. The anonymization process replaces all UIDs, GIDs, and IP addresses in the traces with arbitrary but consistent values. Filenames and paths are anonymized by pathname components: directories are anonymized individually, so that if two paths have a common prefix path, their anonymized forms will share a common prefix as well. In addition, filename suffixes are anonymized separately from the rest of the filename, so all files that share the same suffix will have anonymized names that end in the anonymized form of that suffix.

The anonymization process is configurable – the mapping for the anonymization of any value can be overridden. For example, in the anonymization of our own data, we disable the anonymization of many common file and directory names (such as `CVS`, `.inbox` and `.pinerc`) or components (such as `lock`) and specific UIDs (such as `root` and `daemon`). We also treat several suffixes and prefixes (such as `#`, `,v` and `~`) specially, to preserve the relationship between the anonymized form of filenames containing these suffixes or prefixes and the anonymized form of the filenames without them. It is also possible to configure the anonymizer to omit all filename, UID, GID, and IP information entirely. This makes some analyses impossible, but still provides a great deal of useful information to file system researchers.

Our anonymization approach is vulnerable to a known-text attack by an adversary who has detailed knowledge about the system being traced. For example, an attacker who knows exactly when a particular user accessed a particular file can find the user's anonymized UID and GID and the anonymized filename. Similarly, a chosen-text attack on the system while it is being traced can be used to find the anonymized form of any filename that the attacker can create. Both of these attacks require direct access to or real-time information about the system being traced. An insider can exploit his or her ability to use the file system to gain knowledge about the other users of the system, but an outsider who does not have this leverage cannot reverse the anonymization process. We do not use hashing or any other deterministic method to do the anonymization, because that would allow an attacker to perform a known-text attack without access to the traced system, and it would also allow direct comparison of filenames and other information between traces gathered at different sites.

3 Traced Systems

In order to examine new workloads and place our work in the context of previous studies, we gathered two sets of traces. The EECS trace is reminiscent of the frequently studied computer science departmental workload. The CAMPUS workload is from our university's central computing center and is almost entirely email. The two workloads are summarized in Table 1.

3.1 The EECS System

The main EECS NFS server is a Network Appliance Filer that serves as the primary home directory server for our computer science department. The workload is a mix of research, software development, and course work. EECS users can directly mount their home directories onto their workstations via NFS, Samba, or other protocols. In the EECS system, all non-NFS file access protocols are routed through an intermediate host, which converts these protocols to NFS but prevents us from directly identifying the source of the resulting activity.

Although there is no standard EECS client, a typical client is a UNIX or Windows NT machine with more than 128MB RAM and local copies of system software and most utilities. Most of the EECS clients use NFSv3, but many use NFSv2. All EECS clients use UDP to communicate with the server. The EECS server is used primarily for home directories and shared project and data files. Unlike what we will see in the CAMPUS workload, the EECS traces do not contain any email traffic. In the EECS system, email and WWW service is provided by other servers, and user inboxes are stored on a separate server.

There are no user quotas on the EECS system. The aggregate disk capacity of the EECS client machines is much larger than the capacity of the EECS server. Much of the research data used in the EECS environment is stored on other servers; the only items stored on the EECS server are home directories, data that need to be shared, or data that users want to have backed up. The traces do not include any backup activity.

3.2 The CAMPUS System

CAMPUS is a collection of machines serving the computing needs for the bulk of the administration, college, and graduate school of the University. It handles email and web service for the majority of the students, faculty, and administrators, and has approximately 10,000 active user accounts.

CAMPUS storage is distributed over three NFS servers hosting a total of fourteen 53GB disk arrays. Each NFS server has several network interfaces and they

CAMPUS	EECS
Storage for the campus SMTP, POP and login servers	Storage for the EECS department home directories
Most NFS calls are for data	Most NFS calls are for metadata
Reads outnumber writes by a factor of 3.0	Writes outnumber reads by a factor of 1.4
Peak load periods highly correlated with day of week and time of day	Unpredictable interactive load, but predictable background activity
20% of the files accessed, and 95+% of the data read and written comes from mailboxes	No mailboxes, some mail lock and temporary files
50% of files accessed are mailbox locks	A large number of locks for mail and other applications
Most blocks live for at least ten minutes	Most blocks die in less than one second
Almost all blocks die due to overwriting	Blocks die due to a mix of overwriting and file deletion

Table 1: Characteristics of CAMPUS and EECS.

are configured, via `gated`, to appear as fourteen virtual hosts, one per disk array. Because all NFS traffic to a particular disk array uses an IP address unique to that disk array, we can monitor traffic to the individual arrays.

The connection between the CAMPUS clients and servers is a gigabit Ethernet using jumbo (9000 byte) frames. All clients use the NFSv3 over TCP.

Each of the fourteen disk arrays contains the home directories for a subset of the CAMPUS users. Users are given a default quota of 50MB for their home directories. Users are distributed among the disk arrays according to the first letters of their login names. We gathered long-term traces for two arrays and short-term traces for seven others. We computed summary statistics and general usage patterns for all nine of the traced arrays and found them to be similar. We chose to use the array named home02 for our in-depth analysis.

The central email, WWW, general login, and CS course servers mount the disk arrays via different networks. Our traces capture only the NFS traffic between the email and general login servers and the disk arrays; we do not capture the NFS traffic generated by serving personal home pages or by students working on CS assignments. Statistics provided by CAMPUS sysadmins confirm that the subnet that carries the email and general-purpose login traffic represents the vast majority of the total CAMPUS traffic. The CAMPUS traces do not include backup activity.

Unlike many other systems, mail spools are not kept in dedicated partitions; users' inboxes are located inside their home directories. However, the activity of the CAMPUS system is so dominated by email that in some sense the file systems used for home directories can be considered to be dedicated email partitions. Most users of the CAMPUS system access mail remotely via a POP or SMTP server from their PC or Macintosh. Our traces do not include this POP and SMTP mail delivery activity as such, but do contain records of the NFS traffic this activity generates.

3.3 Data Used

We gathered several months of traces on both systems and then computed summary statistics for the entire trace period and additional analyses for the months of 10/2001 and 11/2001. We observed that the workload for CAMPUS is quite consistent over the entire trace period (once the school schedule, including weekends and holidays, is taken into account). The EECS workload shows considerably more variation from day to day and week to week, but the variance falls within reasonable limits.

For the analyses in this paper, we selected the one-week period of October 21 through October 27, 2001. This week was chosen because it is "typical" in the sense that it is in the middle of the semester and contains no holidays or other unusual events, and because our data for this week contain no gaps. For the EECS system, although each week varies from the others, and none of the weeks appear "average", this week is no more atypical than any other. Each table and graph is labeled with the subset of the data used.

Table 2 shows the summary statistics for the average daily activity of our traces, for both a three month subset and the one week subset used for our in-depth analyses, and compares them to the same statistics from the Baker and Roselli traces [1, 12]. Note that the RES, INS, and NT traces are kernel-level traces of local file systems, and do not show the effect of client-side caching. The Sprite traces, on the other hand, use a different form of client-side caching than NFS.

CAMPUS is an order of magnitude busier than any of the other systems, particularly in terms of the amount of data read and written.

4 Trace Analysis via NFS

Passive NFS tracing is an old idea, and has been used in many trace studies and file system experiments [2, 3, 5, 7, 8]. The technique of passive tracing on a broadcast network evolved in parallel with broadcast net-

	CAMPUS 9/1-11/30	EECS 9/1-11/30	CAMPUS 10/21-10/27	EECS 10/21-10/27	INS	RES	NT	Sprite
Year of Trace	2001	2001	2001	2001	2000	2000	2000	1991
Days	91	91	7	7	31	31	31	8
Total ops (millions)	29.9	2.30	26.7	4.44	8.30	3.20	3.87	0.432
Data read (GB)	135.7	2.27	119.6	5.10	3.05	1.70	4.04	5.36
Read ops (millions)	19.29	0.35	17.29	0.461	2.32	0.303	1.27	0.207
Data written (GB)	45.0	2.94	44.57	9.086	0.542	0.455	0.639	1.16
Write ops (millions)	5.93	0.438	5.73	0.667	0.15	0.071	0.231	0.057
Read/Write bytes ratio	3.01	0.77	2.68	0.56	5.6	3.7	6.3	4.6
Read/Write ops ratio	3.25	0.80	3.01	0.69	15.4	4.27	4.49	3.61

Table 2: A summary of average daily activity during the trace periods. The subset from 10/21 through 10/27 is used for most of our analyses. The INS, RES, NT, and Sprite numbers are from the Roselli and Baker trace studies. INS is an instructional workload, RES is a research workload, and NT is a Windows NT desktop workload.

works such as Ethernet and client/server protocols such as RPC. NFS trace studies began to appear in the literature soon after NFS clients appeared on computer science department networks [7], but they received relatively little attention compared to kernel-based trace studies or studies of other distributed file systems. This changed, however, as NFS became ubiquitous and better tracing tools and methodologies for analyzing NFS traces were devised [2, 8].

Passive NFS tracing is attractive from a research perspective because NFS is a portable and widespread protocol, used in a broad variety of real-world contexts, and so its analysis is applicable to many interesting real-world problems. Unfortunately, we have found that even while computing has become pervasive, it has become increasingly difficult to gather meaningful traces of production servers outside of research labs.

The primary difficulty in gathering traces from production servers is a combination of social, ethical, and legal issues. Detailed file system traces can reveal sensitive information about the activities of organizations or individual users, and the administrators of these systems have justifiable concerns about protecting the privacy of their users. During our data collection, we contacted several commercial ISPs in an effort to gather traces from their sites, and although several of the administrators of these ISPs expressed interest, none of them would permit us to gather traces because of privacy concerns. To address this concern, we added an anonymization step to our tracing procedure, as described in Section 2. This step transforms the traces in such a way that user-specific information such as UIDs or filenames is not revealed, while preserving the information necessary for almost any analysis. We hope that the existence of this anonymizer and the value of our findings will convince ISPs to permit tracing, allow these traces to be shared among the research community, and thereby invigorate contemporary file system design.

A second obstacle to gathering traces is that system administrators are understandably unwilling to modify their systems to allow direct instrumentation because of the risk associated with loading new, non-production, and potentially buggy kernel patches into their critical systems as well as the concern about the additional load it might place on their systems. Passive NFS tracing addresses this concern perfectly; capturing and recording NFS packets from the network requires no changes to the clients or servers and does not introduce any new load on the system.

Despite the decline in the number of new NFS trace studies in recent years, we believe that the technique of passive NFS tracing in tandem with trace anonymization is more important now than ever, because it may allow researchers to trace systems that would otherwise be completely inaccessible.

4.1 Difficulties of Analyzing NFS Traces

As mentioned in earlier NFS trace studies, the analysis of passive NFS traces presents several challenges, particularly when trying to relate these analyses to those done on kernel-based traces [8]. This subsection provides an overview of the challenges of comparing NFS traces with kernel-based traces:

- Details of the underlying file system are hidden.
- The NFS interface is different from the canonical FS interface.
- Client-side caching skews the observed workload.
- Some NFS calls and responses are lost.
- NFS calls may be reordered *en route*.

Just as the idea of passive NFS tracing is not new, methods for dealing with most of these problems are not new, and have been described at length elsewhere [2, 5, 7, 8], and we will discuss these methods only briefly. The problem of reordered calls, however, is something we have not seen addressed in the literature, and so we will discuss our approach in detail in Section 4.2.

4.1.1 The Underlying File System is Hidden

Much information about the underlying server file system is not revealed by any analysis of the messages between NFS clients and servers. For example, it is impossible to learn much about files or directories that are never accessed via NFS, or any information about the actual on-disk layout of the files and directories. It is also impossible to reconstruct the complete file system hierarchy *a priori* – but as shown in earlier studies [2, 8], it is possible to reconstruct the active parts of the hierarchy on-the-fly by learning the relationship between directories and their contents as revealed by `lookup` calls and responses. This method is effective for our traces; after processing several minutes of traces, the probability is very small that we will encounter a file or directory whose parent directory has not already been seen.

Another problem with tracing at this level is that little information about the internal state of the server (such as the state and contents of its cache) is revealed. Because we are interested in characterizing the server workload, which is independent of the internal state of the NFS server, this does not impede our analyses.

4.1.2 The NFS Interface

Analyses that depend on specific details of the file system interface cannot be done on NFS traces. For example, some analyses require knowledge of file `open` and `close` system calls, but neither of these calls exists in NFS. Methods for augmenting a trace with virtual `open` and `close` events are described in related work [2, 8]. We describe our approach for finding *runs* in Section 4.2.

4.1.3 The Effect of Client-Side Caching

The effect of client-side caching is an obstacle to inferring the actual client workload from the workload observed by an NFS server. In order to reduce the latency of client operations, NFS allows clients to cache data and metadata in a weakly consistent manner. This means that some client operations are absorbed by the client cache and never reach the server, or are observed only as `getattr` calls when the client checks whether the data in its cache is still valid.

The strategies employed by some clients can also have the effect that some I/O operations seen by the server might not correspond to actual calls by the client application, but instead are the effect of the client-side operating system performing read-ahead in an attempt to warm its cache with data it anticipates will be requested soon.

The effect of client-side caching and methods to infer actual client workload from the server traffic have been studied extensively [2, 3, 5], but still present new challenges in heterogeneous networks because NFS allows considerable flexibility in how caching is implemented and in the degree of consistency provided to different observers of the file system.

Because our research is primarily concerned with server workloads, which are strongly shaped by client-side caching, we do not want to remove the effects of client-side caching and therefore make no attempt to control for them. This implies that direct comparisons of operation counts and the volume of data read and written between our NFS traces and traces from local file systems should show that the NFS traces contain relatively more metadata traffic (to validate the cache contents) but less data actually read (if the cache is often valid).

4.1.4 Lost NFS Calls/Responses

On the CAMPUS system our monitor consisted of a single gigabit Ethernet port on a fully-switched gigabit network. During bursts of heavy activity, the monitor port simply did not have the bandwidth to forward all of the network traffic. This problem was compounded by the fact that it is impossible to decode an NFS response without seeing the call, so losing a call effectively results in losing both the call and its response. By analyzing the call stream for unexpected holes, and by counting the number of call and responses messages that had no corresponding response or call, we estimate that during period of heavy activity, we lost as many as 10% of the packets through the switch, and for short bursts the percentage could have been even higher. In contrast, on EECS the monitor port was as fast as the port to the server, so we did not observe this problem.

4.1.5 Reordered NFS Calls

Some analyses, such as determining whether a set of accesses are sequential or random, are sensitive to the order of calls. Out-of-order calls occur when NFS calls are delivered to the server in a different order than they were issued by the application. This reordering is largely an artifact of the conventional NFS architecture, in which separate processes, called `nfsiods`, issue the actual network calls. Although a client's calls are dispatched to the `nfsiods` in order, the process scheduler determines the order in which the `nfsiods` run.

To confirm and measure this effect, we performed an experiment using our own clients and server on an isolated network. When the client ran only one `nfsiod`, no call reorderings occurred, but as additional `nfsiods` were added, call reordering became more frequent. In the most extreme case as many as 10% of the packets were reordered, and some calls were delayed by as much as 1 second, although no network errors or packet losses were observed. This effect is more common when UDP is used as the RPC transport, but can also occur with TCP.

4.2 Detecting Runs in NFS

Earlier trace studies [1, 9, 12, 15] describe workloads in terms of sequential runs. The notion of a sequential run is important to the heuristics that file systems use to efficiently process a stream of requests to a file. For example, FFS prefetches blocks when an access pattern appears sequential and does not when it appears non-sequential. Determining a client's underlying access pattern is an important part of optimizing file system access.

In order to compare our traces to earlier studies, we needed a method to divide NFS records into runs on a file. When we applied the conventional analysis of considering all consecutive accesses to a single file as a sequential run, we obtained a much higher percentage of random accesses than previous work had led us to expect. We determined that the apparent randomness was due to two factors: the reordering of requests introduced by multiple client-side `nfsiod` processes and the omission of a small percentage of NFS records from our logs. Because of these factors, the conventional methods are ineffective for NFS analysis.

Previous studies define a *run* as the series of accesses to a file between each `open` and subsequent `close`, and an *access* as either a single read or write. Because `open` and `close` do not exist in NFS, we use the following methodology to create runs:

1. We associate a list of accesses with each file, adding a record to this list whenever we see a `read` or `write` to the file in the trace.
2. We then convert this list into a collection of one or more runs.
   (a) If the last item referenced the end-of-file, begin a new run.
   (b) If the last item in a list is old (*e.g.*, older than 30 seconds) begin a new run.
   (c) Else, add the current item to the current run.

Using this mechanism, a series of accesses can be split into one or more runs. We also evaluated other methods to break lists of accesses into runs. For example, a break might occur when a gap of several seconds occurs between two accesses or when a large backwards seek occurs. We found that these other methods produced inconsistent results.

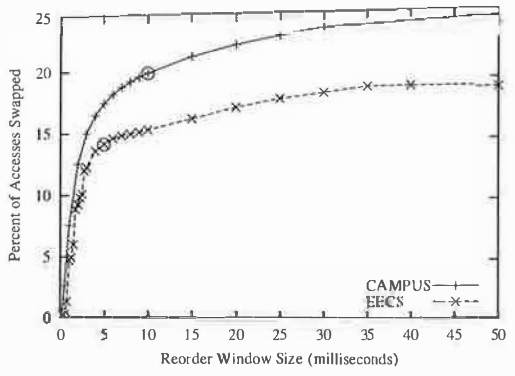


Figure 1: The effect of different window sizes on the percentage of swapped accesses. The sorting window size chosen for each host is shown with a circle: for EECS, 5ms is sufficient, but for CAMPUS 10ms is better.

After splitting accesses into runs, we separately determined each run's access pattern. Informally, a run is sequential if each request begins where the previous one left off, for all requests. In NFS terms, a run is sequential if, for all accesses in the run, $\text{offset}_i = \text{offset}_{i-1} + \text{count}_{i-1}$, where i is the index of the current access in a run and count_{i-1} is the number of bytes accessed in the previous access. Offsets and counts are rounded up to blocksizes of 8k. For example if a series of offset(counts) was 0k(8k), 8k(8k), 16k(7k), 24k(8k), we would consider this series to be sequential despite the missing 1k between the third and fourth requests. If a run has the qualities of being sequential and also accesses the file from offset 0 through to *eof*, it is called *entire*. If it is not sequential, the run is called *random*. A run is called "read" if it contains only read requests, "write" if is contains only writes, and "read-write" if it contains one or more of each.

If we do nothing to compensate for the reordering that occurs due to `nfsiod` scheduling, we observe an unnaturally large percentage of random accesses. In order to avoid this phenomenon, we partially sort requests in ascending order within a small temporal window, which we call a *reorder window*. For each request, we look ahead t milliseconds to see if a nearby request should be swapped with the current request, and swap them if they are out of order. We show the results of using differing *reorder window* sizes on a subset of the data (Wednesday 10/24/2001, 9am-12pm) in Figure 1.

Note that we want to use the minimum sorting window that removes the reordering effect of the `nfsiods` but does not mask true client randomness – with an infinite sorting window, any workload that visits every block of a file in any order will appear sequential. Both workloads show a great improvement in observed sequentiality with a sorting window of only a few milliseconds and then exhibit a knee, after which larger sorting windows give much smaller gains in the observed sequentiality. Empirically, the results from this analysis suggested that we should use a window size of $t = 5$ milliseconds for EECS and $t = 10$ milliseconds for CAMPUS. For the remainder of this paper, the sequentiality analysis includes this sorting step, unless otherwise noted.

5 Comparison With Previous Traces

In this section, we place our traces in the context of several often-cited traces from earlier studies. We examine run patterns and block and file lifetimes. In Section 6, we discuss analyses specific to our traces.

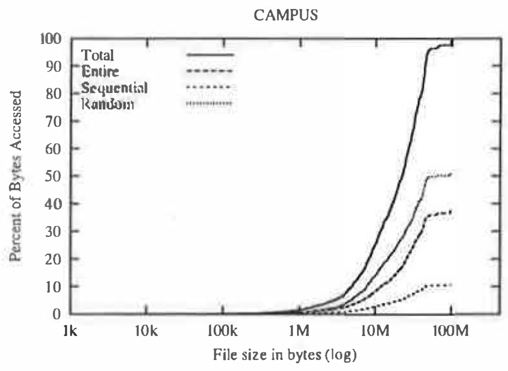
5.1 Run Patterns

Most previous studies categorize workloads by their level of sequentiality, because many file system layout optimizations rely upon some degree of sequentiality.

The rightmost columns of Table 3 compare EECS and CAMPUS to three historical traces, using the heuristic described in Section 4.2 to find the runs in our traces. Both EECS and CAMPUS have significantly different run patterns than earlier traces. Both workloads contain a much higher percentage of write runs (especially EECS, where write operations already outnumber read operations in the overall operation count). We initially believed that much of the EECS write activity is due to late night batch jobs, but examining only peak hours revealed essentially the same percentages. Peak hour analysis for CAMPUS also yields the same percentages shown in Table 3. In EECS, we believe that the dominance of write traffic is because most client machines are used by one user and that user's files experience little cache invalidation. In CAMPUS, the read-write imbalance has a different cause: most writes are short appends to mailbox files, and most reads are long reads of mailbox and other mail-related files.

Note that the classification of "random" and "sequential" used in Table 3 follows the heuristics used by many file systems, which categorize all non-sequential access patterns as random. Therefore highly regular access patterns, such as stride access patterns or reverse scans, would be overlooked by this classification. A visual inspection of the non-sequential access patterns in our traces did not reveal a significant number of accesses that had any discernible pattern other than sequential sub-accesses separated by seeks.

Figure 2 illustrates the file size-based access patterns of EECS and CAMPUS. EECS is similar to previously



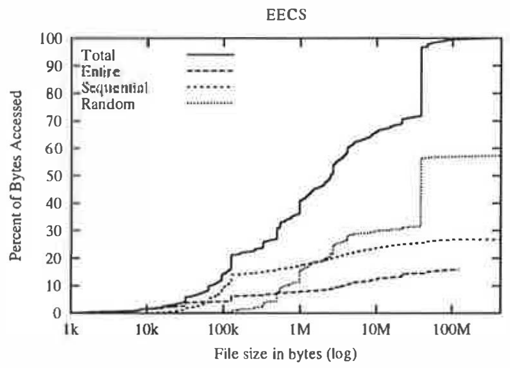


Figure 2: Percentage of bytes accessed randomly, sequentially, or in their entirety verses the total bytes accessed. Each run is categorized according to its access pattern and then all of the bytes accessed in this run are added to the subtotal for this category. These runs are computed by the heuristic described in Section 4.2.

analyzed file systems where a large percentage of accesses come from files smaller than 1M. Almost 60% of bytes are accessed randomly, which is about the same as Roselli's NT workload but much larger than most other workloads. Again like the NT workload, long files read in their entirety constitute 30% of the total in EECS. CAMPUS is similarly dominated by random and entire runs, although as discussed in Section 6.4 most of these runs that are categorized as "random" do contain long, completely sequential sub-runs, and should be considered highly sequential.

The vast majority of CAMPUS bytes transferred come from files larger than 1M. This is unlike almost all of the previous work on run analysis, except for the two outlier traces of the Baker study [1].

Previous analysis, Roselli in particular, showed that the majority of bytes from files over 100k are accessed randomly. As mentioned in Section 4.2, our data indicates that "random" is too strong a term and too coarse a measurement. In our traces, most runs that would be categorized as random in the entire/sequential/random

Access Pattern	CAMPUS	EECS	NT	Sprite	BSD	CAMPUS	EECS
	Raw					Processed	
Reads (% total)	53.1	16.6	73.8	83.5	64.5	53.1	16.5
Entire (% read)	47.7	53.9	64.6	72.5	67.1	57.6	57.2
Sequential (% read)	29.3	36.8	7.1	25.4	24.0	33.9	39.0
Random (% read)	23.0	9.3	28.3	2.1	8.9	8.6	3.8
Writes (% total)	43.8	82.3	23.5	15.4	27.5	43.9	82.3
Entire (% write)	37.2	19.6	41.6	67.0	82.5	37.8	19.6
Sequential (% write)	52.3	76.2	57.1	28.9	17.2	53.2	78.3
Random (% write)	10.5	4.1	1.3	4.0	0.3	9.0	2.1
Read-Write (% total)	3.1	1.1	2.7	1.1	7.9	3.0	1.1
Entire (% r-w)	1.4	4.4	15.9	0.1	NA	3.5	5.8
Sequential (% r-w)	0.9	1.8	0.3	0.0	NA	2.1	7.3
Random (% r-w)	97.8	93.9	83.8	99.9	75.1	94.3	86.8

Table 3: File access patterns using entire/sequential/random categorization. The leftmost two columns show CAMPUS and EECS traces divided into runs according to the sorting mechanism described in Section 4.2, but otherwise unaltered. The rightmost two columns show the CAMPUS and EECS traces divided into runs using the complete methodology presented in Section 4.2. to account for request reordering produced by `nfsiods` and prevent small seeks (of less than 10 8k blocks) from changing a run from sequential to random. In both this presentation and that in section 6.4, singleton runs are either sequential (if they access only part of the file) or entire (if they access the entire file). The third, fourth, and fifth columns show the results from Roselli's NT, the Sprite, and the BSD studies, respectively.

taxonomy are actually composed primarily of sequential sub-runs separated by short seeks. We present a new metric for run sequentiality in Section 6.4.

5.2 File and Block Lifetimes

The distribution of block and file lifetimes is a workload characterization that can be exploited by the file system. If blocks do not live long, it may be possible to keep them in memory and avoid ever writing them to disk. If the block lifetimes are bimodal, then once a block lives sufficiently long, it might be beneficial to optimize its on-disk layout under the assumption that the block will live for a long time and the cost of reorganization can be amortized across many future accesses.

We used Roselli's "create-based" method [12] to calculate block lifetime statistics. We processed our data in two separate phases. In Phase 1, we record both block births and deaths. In Phase 2, the *end margin* of Roselli, we record only block deaths. To remove sampling bias for blocks born early in the first phase, we remove any death records for blocks with lifespans longer than the length of the second phase. All blocks whose lifetimes exceed the length of Phase 2 are considered *end surplus*.

5.2.1 Analysis

We ran a set of five 24-hour block life analyses on CAMPUS and EECS for the weekdays in our trace period. The first phase of each test began at 9am and ran for 24 hours with a 24-hour end margin. We chose a 24-hour end margin because shorter margins did not capture the relatively high percentage of deaths that occur between 18 hours and 24 hours. Longer tests agree with earlier observations that a block that lives for a day is likely to live for a relatively long time. The daily end surplus ranges between 2.1% and 5.9% for CAMPUS and between 3.5% and 9.5% for EECS. Logically, this daily surplus must eventually be balanced by deletions (via periodic purges when disk quotas are reached or the file system fills up, or more gradual and steady deletion), because otherwise CAMPUS and EECS would be filled in a matter of weeks. We have not investigated this phenomenon, however.

5.2.2 Summary of Block Births and Deaths

The summary statistics for block deaths and births are shown in Table 4. Both CAMPUS and EECS exhibit similar trends with respect to block births. In both systems, most births are due to actual data writes, as opposed to preallocation of space (*e.g.,* using `lseek` to lengthen a file). In fact, the number of file extensions is mildly exaggerated, because writes that follow an `lseek` past the end-of-file are interpreted as extension writes to all the newly created blocks; that is, not only the blocks explicitly written, but all unwritten blocks between the previous end-of-file and the new block are counted as extensions. In the future it might be useful

	CAMPUS	EECS
Total Births (millions)	28.4	9.8
Due to Writes	99.9 %	75.5 %
Due to Extension	≪ 0.1 %	24.5 %
Total Deaths (millions)	27.5	9.2
Due to Overwrites	99.1 %	42.4 %
Due to Truncates	0.6 %	5.8 %
Due to File Deletion	0.3 %	51.8 %

Table 4: Daily block life statistics for 10/22-10/26/2001. Note that only the block deaths that occur within 24 hours of the block birth are reported here, and so the total number of blocks that are born and die within this five day period is higher than the **Total Deaths** figure.

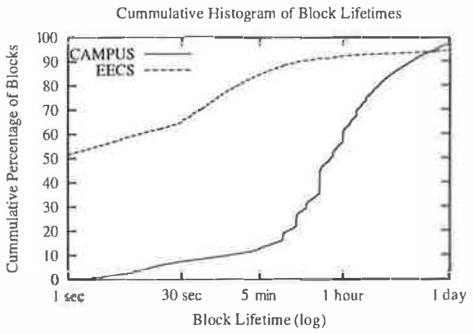


Figure 3: The cumulative distribution of block lifetimes for each day 10/22-10/26/2001.

to distinguish between these cases. Even with this overcounting, however, relatively few blocks are created via file extensions, particularly on CAMPUS.

Earlier studies found that most blocks die due to overwriting. In the Roselli trace results [12], the percentage of blocks that die due to overwriting is always greater than 50% and often between 85%-99%. Our results follow this pattern, but show an important difference between CAMPUS and EECS. On CAMPUS, more than 99% of the blocks die due to overwrites. For CAMPUS almost all the bytes written are to mailboxes, which are never deleted but are overwritten frequently.

The distribution is more varied on EECS. There are considerably fewer overwrites on EECS than CAMPUS, and many more removes. This is due to the research-oriented workload on EECS: tasks such as compiling programs, using source control tools, and manipulating data can create and delete many temporary files. There are also approximately 10,000 deletes per day of small files with names of the form "`Applet_*_Extern`", which are files created by UNIX window managers. Web browser caches and email composition files make up most of the rest of the deleted files.

On CAMPUS, more than 96% of the files created and deleted during the course of a day are zero-length lock files. For EECS, lock files account for only 8% of the files created and deleted each day. Since these are zero-length files, however, they do not consume any data blocks, and so this does not have an effect on the block lifetime statistics.

5.2.3 The Lifespan of Blocks

The lifespan of a block is defined as its time-of-death minus its time-of-birth. Figure 3 illustrates that the two systems are quite different. For EECS, over half the blocks die in less than a second and relatively few blocks live for an entire day; this is similar to the results of the NT study by Vogels [15]. Most of the blocks that die in less than one second on EECS belong to log or index files that are written frequently and in an unbuffered manner.

On CAMPUS, blocks tend to live longer; about half live longer than 10-15 minutes. These results are more reminiscent of the results of Baker [1] than they are of the more recent studies by Vogels [15] and Roselli [12], although the curve of our distributions is similar.

On CAMPUS, blocks live longer because many are born due to mail delivery (at the end of the mailbox) or saving the mailbox before exiting an email client, and blocks die when mail messages are removed from the mailbox, a process that most email clients do in a batch operation. Thus we expect many blocks to live for approximately the same length of time as an average email session. Although we do not know this length, our data suggest mail-reading session times typically range between fifteen minutes and an hour.

In comparison with Roselli's block lifetime analysis results [12], we see that our results are quite different. The only similarity we observed was the fact that both our CAMPUS trace and Roselli's RES trace have a knee at approximately the 10-minute mark. However after this 10-minute mark, CAMPUS has a more gradual increase of block lifetimes, whereas for RES the change is abrupt. On CAMPUS, few blocks live for less than a second. Approximately 50% and 20% die within a second for EECS and Roselli traces, respectively. The end surplus, those blocks that live longer than a day, is about 5%-7% for both CAMPUS and EECS. For Roselli, this percentage varies between 10%-30% for non-NT traces and is about 70% for the NT traces.

6 New Findings

In this section, we discuss observations specific to our traces and to the EECS and CAMPUS workloads.

6.1 Characterizations of EECS and CAMPUS

EECS and CAMPUS differ on a fundamental level. EECS is dominated by writing and by metadata queries, particularly client queries about whether their cached copies of the EECS data are still valid. The CAMPUS workload, in contrast, is dominated by reading and writing large files and by creating and deleting lock files.

CAMPUS is utterly dominated by email and the daily rhythms of user activity. EECS is similar to the research workloads already documented in the literature, but has a lower read/write ratio.

We observe that applications frequently use the file system for ephemeral data storage and for locking. These operations can impose a significant load on file systems, especially when these temporary, cache, or lock files are actually hosted on a different machine than the application. Either file system designers should be sure that systems can identify and handle these cases efficiently, or application designers should be encouraged to use more suitable metaphors for locking and temporary storage.

Our results also provide more evidence that, as shown in previous studies, delayed writes can substantially reduce the amount of actual writing done by the file system, because many blocks do not live long enough to be written. This is particularly true for data blocks on EECS. We also believe that this will be true for the metadata blocks on CAMPUS because of the incessant creation and deletion of short-lived zero-length lock files.

6.1.1 The EECS Workload - Research

The EECS workload is predominantly file attribute calls (`lookup`, `getattr`, and `access`). Most of these calls occur because clients are simply checking to see whether a file has been updated or whether they can use a cached copy. Unlike CAMPUS, which is read-oriented, the overall EECS read/write ratio is 0.69, but varies widely over time, with periods of heavy read activity. This is illustrated in Figure 4. Note that the average hourly read/write ratio shown in Table 5 is skewed by periods of relative inactivity that are dominated by reading, and hence the hourly average is quite different from the overall average.

Somewhat perversely, much of the EECS workload is caching web pages viewed by users running on client workstations. By default, these browser caches are created in a subdirectory of the user's home directory, so they are "cached" on the central file server instead of locally on each machine. Similarly, many of the most frequently created and deleted files on EECS are files created by the window managers and desktop applications of some users (for example, `Applet` files created by GNOME).

Aside from the central storage of the web pages and `Applet` files, if our EECS workload is an instance of the "typical departmental server," then not much has changed in the past fifteen years. Our traffic patterns resemble the model postulated by Ousterhout *et al.*[9], which predicted that as cache sizes grow, most reads will be serviced from cache and write performance will become the bottleneck. This prediction was the motivation for several new developments in file system design, such as LFS [13]. Ousterhout's conjecture that we may eventually be able to replace rewritable media such as magnetic disks with less expensive and write-only storage devices (tested successfully for the Venti system [11]) does not appear to hold for EECS or CAMPUS because even though cache sizes have increased since 1985, the size of file systems and the amount of data accessed by applications has also grown.

We speculate that the NFSv4 lease and delegation mechanisms [10] could eliminate a large fraction of the NFS calls generated by the EECS workload by removing many of the situations where a client is contacting the server simply to confirm that its cached copy of a file is up-to-date.

6.1.2 The CAMPUS Workload - Email

On CAMPUS, email is the dominant cause of traffic. During the peak load hours, about 20% of the unique files referenced are user inboxes, and another 50% are lock files used to control concurrent access to these inboxes. Many CAMPUS users use email applications that access additional configuration files, put incoming email into other mailboxes, and create temporary files to hold messages during composition. We do not identify each of these kinds of files here, but we have observed that a large fraction of the remaining accessed files are also related to email use.

These numbers and our analysis of the traces support the hypothesis that most CAMPUS users do little else on the system besides use email. A typical user session involves logging in (accessing `.cshrc` and possibly `.login`) and starting an email client. The email client typically reads a configuration file, creates a lockfile for the mailbox, and then scans the mailbox file. During a mail session, mail applications may rescan the mailbox several times. Composing email messages may, depending on the mail program, create temporary files in home directories, and viewing or extracting attachments may also create files. Quitting the mail client causes some or all of the mailbox file to be rewritten.

Even more dominant is the contribution of email to the total quantity of data movement. For both the total

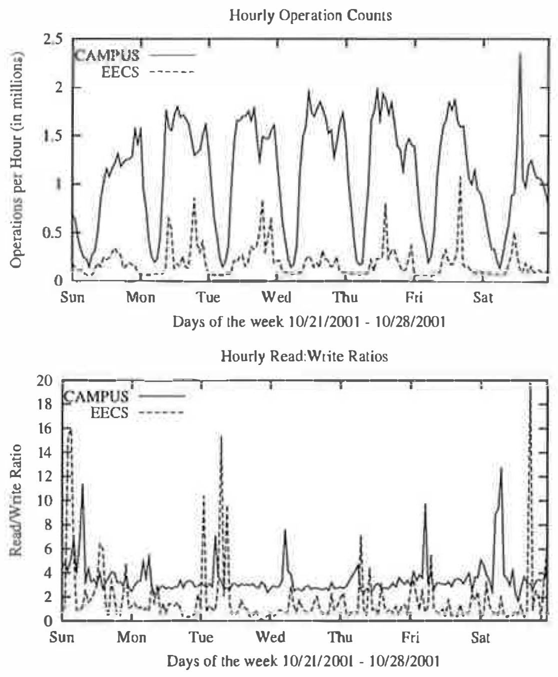


Figure 4: Variation of the hourly total operation count and read/write ratios for the week of 10/21/2001.

	All Hours			
	CAMPUS		EECS	
Total Ops (1000s)	1113	(48%)	185.1	(86%)
Data Read (MB)	4989	(45%)	212.3	(165%)
Read Ops (1000s)	719	(48%)	19.7	(110%)
Data Written (MB)	1856	(58%)	378.5	(246%)
Write Ops (1000s)	239	(58%)	28.6	(201%)
R/W Op Ratio	3.27	(48%)	3.16	(242%)
	Peak Hours Only			
	CAMPUS		EECS	
Total Ops (1000s)	1699	(7.6%)	267	(68%)
Data Read (MB)	7153	(6.1%)	268	(146%)
Read Ops (1000s)	1088	(7.1%)	29.2	(77%)
Data Written (MB)	2934	(12%)	439	(228%)
Write Ops (1000s)	377	(12%)	341	(158%)
R/W Op Ratio	2.46	(10%)	1.13	(106%)

Table 5: Average hourly activity. The *All Hours* columns are for the entire week of 10/21-10/27/2001. The peak hours are the hours 9am-6pm, Monday 10/22/2001 through Friday 10/26/2001. The numbers in parentheses are the standard deviations of the hourly averages, expressed as a percentage of the average.

number of read and write operations and the total volume of data transferred, more than 95% of the data read and written involve a user's primary inbox.

One of the causes of the large read load on CAMPUS is an unfortunate interaction between NFS's file-based caching model and the flat-file inbox and mail clients used on CAMPUS. Delivering a message to an inbox updates the modification time on the entire file, even though only a small number of blocks might actually have been changed. For a typical inbox on CAMPUS this results in the invalidation and immediate re-reading of, on average, more than 2 megabytes of data in the client cache. This component of the workload represents the majority of all reads on CAMPUS. We speculate that if client caching of mailboxes was done on a block or message basis instead of a file basis, the amount of data read per day would shrink to a fraction of the current size.

6.2 Variance of the Workloads due to Time

Like Vogels [15], we observe extremely large variance of load characterization statistics over time. We also observe, however, that much of this variance can be explained by high-level changes in the workload over time. This correlation has been observed in many trace studies, but its effects are usually ignored.

The most notable change in our traces is the difference between peak and off-peak hours, where peak hours are 9am-6pm on weekdays. Figure 4 illustrates the cyclical pattern of the CAMPUS load. It also portrays the consistent read/write ratio during peak hours and its tendency to spike during off-peak hours, when a few accesses can skew the ratio. Table 5 quantifies the reduction in normalized variance during peak hours, conveying that time is a strong predictor of operation counts, amount of data transferred, and the read-write ratio for CAMPUS.

We examined a range of possibilities for the "peak" hours for CAMPUS and found that using 9am-6pm resulted in the least variance. The standard deviation, expressed as a percentage of the mean, is reduced by a factor of at least 4 for all of the CAMPUS statistics when only these peak hours are considered. The same peak hours were also those that resulted in the least variance for EECS, although there is less correlation between the EECS workload and the "regular" work week. In many cases, the load spikes from Figure 4 are directly attributable to specific causes, such as software builds, large experiments, or data processing, which are often run via `cron` during off-hours.

Understanding the time-varying nature of a workload is essential for any kind of dynamically-optimizing storage system. Systems that experience genuine quiet periods can use them to rearrange data in anticipation of the next period of heavy use. Less radically, if the system is designed to make file layout decisions based upon workload, it is useful to know that there are atypical periods (for example, when backups are running) that might best be ignored, in order to avoid optimizing for an uncommon or non-critical workload.

6.3 Predicting File Attributes via File Names

One purpose of our data collection was to explore ways of predicting future file access patterns in order to optimize file and block layout. Cao *et al.* propose that applications can provide hints to the file system about what the application believes the future properties and access patterns of their files will be [4]. This method has not been widely adopted because it requires modifying the applications to provide these hints. However, we found that on CAMPUS and EECS, applications do provide hints in that the filenames chosen for new files are accurate predictors of the lifespan, size, and access patterns of nearly all files. We also observe that file renames are rare, which means that these hints are rarely wrong or need to be modified. This means that the file system has, at the time of file creation, reliable and potentially useful information to guide its decisions.

On CAMPUS we can predict the size, lifespan, and access patterns of most files extremely well simply by examining the last component of the pathname. Nearly all of the files on CAMPUS fall into one of the four categories: lock files, dot files, mail composer files, and mailboxes, and the size, lifespan, and access patterns are predicted strongly for each of these categories.

Zero-length lock files make up 96% of the files that are both created and deleted during our test week, and 99.9% of these lock files live less than 0.40 seconds. Temporary files created by the mail composer account for 2.5% of the files created each day; 45% of these live less than 1 minute, 98% are less than 8K in length, and 99.9% are smaller than 40K. Most dot files fit in one block, although there are some multi-block dot files (for example, the primary mail client configuration file `.pinerc` varies in size from 11K to 26K). The mailbox files are considerably larger than any other commonly-accessed file and are never deleted.

Activity on EECS is a union of several kinds of activities, and the combined workload is more complex than CAMPUS. However, our preliminary analyses show that for most files on EECS, the pathname of a file is also a strong predictor of file attributes. Analyzing these relationships and developing methods for the file system to infer and exploit them is the subject of ongoing work.

6.4 Run Patterns and Sequentiality

In order to quantify the degree of sequentiality in a run to a finer level than simply sequential or random, we introduce a *sequentiality metric*, based on Keith Smith's *layout score* [14]. Our *sequentiality metric* is the fraction of blocks that have been accessed sequentially. A block is accessed sequentially if it is consecutive to the previous access. A run with a sequentiality metric close to 1.0 is almost sequential and should be processed by the file system and disk as if it were.

To account for the minimal penalty introduced by small jumps, we introduce the term *δ-consecutive* to mean that a block is within δ blocks of its predecessor. In our analyses we have arbitrarily chosen a δ of 10 blocks; logical jumps of less than 10 blocks are unlikely to require seeks, but these jumps also account for a large percentage of the jumps we observed.

Figure 5 examines the differences in average sequentiality metrics for different run lengths. Long reads on CAMPUS are typically highly sequential. Long CAMPUS writes, however, tend to touch several sequential blocks and then seek to a new location, either forward or backward in the file. This is reflected in a metric of about 0.6, which means that only 60% of the accesses are δ-consecutive. Long EECS reads also tend to exhibit highly sequential behavior, although not to as great a degree as CAMPUS. The jaggedness of the plot of long EECS runs is caused by the scarcity of occurrences of runs of this size in EECS and the high variance between those occurrences. EECS writes still tend to be highly-seek prone, often having average sequentiality metrics below 0.5, even when $\delta = 10$.

The central difficulty in the results produced through the mechanisms of partial reordering (described in Section 4.2) and allowing small jumps is that they only enable approximations of the clients' behaviors. If a series of requests is genuinely out-of-order or makes small jumps, these two mechanisms will obscure this behavior. However, an NFS server must take into account client activity when deciding to prefetch. That requests are reordered despite the best efforts at the client to issue them in an optimal order suggests that a more intelligent algorithm must be used by the server to optimize accesses.

To investigate the effect of reordered requests on NFS read performance, we performed an experiment by modifying the FreeBSD 4.4 NFS server to employ a simplified version of the sequentiality metric presented here in its read-ahead heuristic. On a loaded system, we observed that nearly 10% of the requests were reordered, and in this situation our new heuristic improved end-to-end transfer speed for large sequential transfers by more than 5%.

The need to apply heuristics to adequately support client-side access patterns and the results of our experiment suggest that NFS servers must be intelligent in categorizing a client's access: a single out-of-order access should not relegate it to the "random" dustbin. In our traces, the vast majority of seeks were to blocks two or three away from the current offset. These may have been due to lost packets or may be a genuine reflection of the

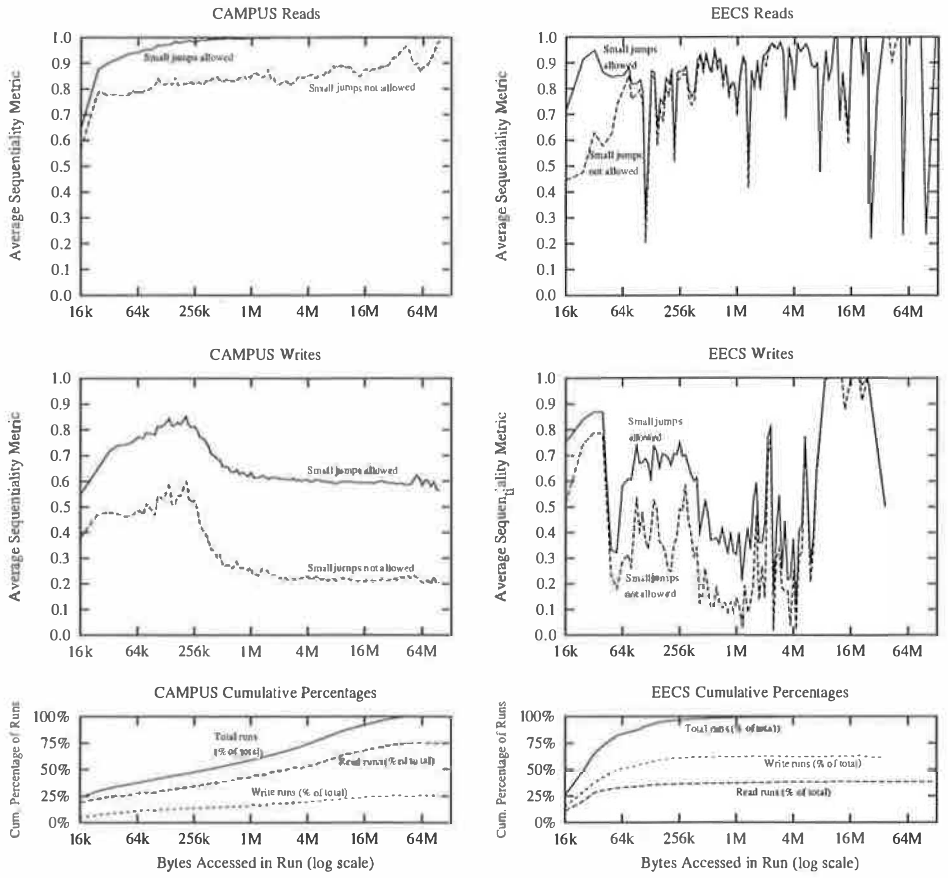


Figure 5: Bytes Accessed vs. Sequentiality Metric. $\delta = 10$ shown as "small jumps allowed"; $\delta = 0$ as "small jumps not allowed."

access stream, but in either case runs containing only adjacent and nearby requests are served more efficiently if they are considered sequential. In our categorization, we consider any jump of fewer than 10 blocks sequential, because on today's disks, if the file is laid out contiguously on disk, then logical seeks of fewer than 10 blocks are unlikely to induce disk arm movement.

7 Conclusion

Some of our results strengthen or support the heuristics behind contemporary file system design, while others portray new aspects of NFS workloads that should be taken into consideration in the design of future file systems and NFS servers. Our analysis of our new traces also suggests several new research questions.

- We find that NFS servers must be prepared to deal with reordered calls. If they use fragile metrics to estimate the sequentiality of an access pattern, reordered calls may degrade their performance.
- Optimizations based on file pathnames could assist servers in optimizing their file layout. The predictions are not uniform across systems, but must be learned from observations of each system. The predictions are particularly accurate on single-application systems, such as mail servers, which are becoming more common.
- Our traces show a continuation in the trend that in traditional computer science workloads, many blocks die quickly. We also see this in email workloads. Mechanisms for delaying writes, such as NVRAM, would improve performance for both the CAMPUS and EECS workloads.
- Most long read runs that conventional metrics label as "random" actually contain long, entirely sequential sub-runs, so servers that recognize these somewhat more complex patterns can improve performance by optimizing the files for sequential access.
- Although long write runs exhibit higher variance in sequentiality than long read runs, 60% of their block accesses are sequential.
- On EECS and especially CAMPUS, servers could schedule periods of reorganization since the daily

and weekly pattern of the workload is predictable.

- While it may have been obvious *a priori*, flat-file mailboxes are quite inefficient. Anecdotal evidence and recent experiments suggest that database-driven mail servers can be faster and consume fewer resources than file-system based servers [6]. It might be possible to build a file system to serve mail loads as efficiently as database-driven mail servers, but it is not clear whether such a file system offers any compelling advantages.

Our findings suggest a number of ways that file servers can optimize their performance by analyzing the workload they observe, but it remains to be seen whether the benefit of those optimizations is worth the cost. For example, we do not know how much data and computation are necessary for a general purpose file system to derive and take advantage of the strong correlation between filenames and file size or lifespan. The larger question is whether it makes sense to attempt this optimization at all, or instead simply accept the fact that general purpose file systems can never perform as well as specialized ones, and focus our efforts on designing file systems for specific workloads such as mail service, WWW service, the desktop, and other new workloads as they emerge.

8 Acknowledgments

We could not have collected our traces without generous help from the administrators of each system, particularly Alan Sundell and Bill Ouchark of CAMPUS and Peg Schaefer and Aaron Mandel of EECS. The paper benefited enormously from the thoughtful comments from our reviewers and Remzi Arpaci-Dusseau, our paper shepherd. This work was funded in part by IBM.

9 Obtaining a Copy of Our Traces

Researchers interested in acquiring a copy of our tools for collecting anonymized traces, or a copy of the anonymized copies of the traces described in this paper (or newer traces, as they become available) should contact the authors at *sos@eecs.harvard.edu*. We are in the process of constructing a public repository for traces and related tools, and are actively gathering contributions from other researchers.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Monterey, CA, October 1991.

[2] Matthew A. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, San Fransisco, CA, January 1992.

[3] Matthew A. Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[4] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–177, Monterey, CA, November 1994.

[5] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Monterey, CA, 1994.

[6] Nicholas Elprin and Bryan Parno. An Analysis of Database-Driven Mail Servers. Technical Report TR-02-13, Harvard University DEAS, 2002.

[7] Riccardo Gusella. The Analysis of Diskless Workstation Traffic on an Ethernet. Technical Report 379, University of California at Berkeley, 1987.

[8] Andrew W. Moore. Operating System and File System Monitoring: a Comparison of Passive Network Monitoring with Full Kernel Instrumentation Techniques. Master's thesis, Monash University, 1995.

[9] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, WA, December 1985.

[10] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, , and R. Thurlow. The NFS Version 4 Protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE2000)*, Maastricht, The Netherlands, May 2000.

[11] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *First USENIX Conference on File and Storage Technologies*, pages 89–102, Monterey, CA, 2002.

[12] Drew Roselli, Jacob Lorch, and Thomas Anderson. A Comparison of File System Workloads. In *USENIX 2000 Technical Conference*, pages 41–54, San Diego, CA, 2000.

[13] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[14] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.

[15] Werner Vogels. File System Usage in Windows NT. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island, SC, December 1999.

Modeling Hard-Disk Power Consumption

John Zedlewski* Sumeet Sobti* Nitin Garg* Fengzhou Zheng*
Arvind Krishnamurthy† Randolph Wang*

Abstract

Excessive power consumption is a major barrier to the market acceptance of hard disks in mobile electronic devices. Studying and reducing power consumption, however, often involves running time-intensive disk traces on real hardware with specialized power-monitoring equipment. This paper presents Dempsey, a disk simulation environment that includes accurate modeling of disk power consumption. It includes tools to automatically extract performance and power consumption parameters from a given disk drive, without needing detailed specifications from the manufacturer. The tools use stimulus-based measurements to extract these parameters. Dempsey is experimentally validated for two mobile hard disks, namely, the 1 GB IBM Microdrive and the 5 GB Toshiba Type II PC Card HDD. In the worst observed case, Dempsey's estimate of power consumption differs from the measured consumption by 7.5%. This demonstrates that disk power consumption can be simulated both efficiently and accurately.

1 Introduction

Many mobile electronic devices, such as MP3 players, digital cameras and personal digital assistants, exhibit an almost insatiable demand for storage capacity. These devices have traditionally relied on compact flash memory, which is reasonably fast and power-efficient, but also expensive and capacity-limited. Recent advances in magnetic disk technology have made possible the development of high capacity, small form-factor disk drives that are compatible with traditional mobile interfaces, such as PCMCIA and CF+ Type II slots.

*Department of Computer Science, Princeton University. Email: zedlwski@princeton.edu, {sobti, nitin, zheng, rywang}@cs.princeton.edu.

†Department of Computer Science, Yale University. Email: arvind@cs.yale.edu.

This work is supported in part by IBM. Wang is supported by NSF Career Award CCR-9984790 and Krishnamurthy is supported by NSF Career Award CCR-9985304.

Unfortunately, one important hurdle continues to block the widespread acceptance of these miniature hard disks in mobile devices: *power consumption*. Recent studies have demonstrated that a small form-factor disk, such as the IBM Microdrive, may consume 5-10 times more power than its flash memory counterparts [26]. In the context of a notebook computer with a powerful lithium-ion battery, these levels of energy consumption are quite bearable, and the storage subsystem in such computers is rarely responsible for more than 10-30% of the overall power drain [4]. In an MP3 player running on AAA batteries, on the other hand, every Joule is critical. Thus, effective power management of hard disks, especially the mobile ones, is becoming increasingly important.

Research in effective disk power management can be a frustrating process, as meaningful disk traces take days to run, and researchers need access to expensive power-monitoring equipment. In this scenario, simulation quite naturally seems like an attractive approach to follow. Simulation software has already been demonstrated to be able to model disk performance, both efficiently and accurately [6, 19, 20]. In this paper, we present Dempsey (Disk Energy Modeling and Performance Simulation Environment), a tool that seeks to bring a similar level of accuracy and convenience to energy-aware storage system designs.

Dempsey extends the well-tested DiskSim simulator [6] to model *power* consumption in addition to *performance* characteristics of hard disks. It includes a set of tools to extract the necessary power consumption parameters from a given disk. Dempsey uses stimulus-based measurements to derive all the required performance and power consumption parameters. Thus, no detailed specifications from the manufacturer are necessary. This enables it to handle disks with the IDE interface, which in general lacks commands to determine specific internal parameters of the disk. This ability to handle IDE disks is significant, since IDE is the dominant standard for mobile disks today.

Dempsey is experimentally validated for the

1 GB IBM Microdrive and the 5 GB Toshiba Type II PC Card HDD using a variety of synthetic and real-world traces. For the IBM Microdrive, Dempsey's estimate of power consumption differs from the measured consumption by 7.5% in the worst observed case. In the average case, however, the error is only 1.8%. The corresponding errors for the Toshiba HDD are 6.9% and 3.6% respectively. On a modern desktop machine, Dempsey is able to simulate traces at a rate of more than 8000 disk-requests per second.

Section 2 provides an overview of relevant existing work in the fields of disk performance modeling and disk power management. Details of Dempsey's design and implementation, including the performance and power modeling components, are given in Section 3. In Section 4, experimental results are presented, which validate the simulator for the two representative disks. Section 5 presents the conclusions.

2 Related Work

Dempsey attempts to connect two research fields: *disk performance modeling* and *disk power management.* We survey some related work in these two fields.

2.1 Disk Performance Modeling

Ruemmler and Wilkes present a thorough introduction to disk performance modeling [19], and convincingly demonstrate the need for sophisticated disk-simulation techniques. They suggest the use of the "demerit figure" as a measure of a simulator's accuracy. The demerit is defined as the root mean square of the horizontal difference between the simulated and real response-time distribution curves for a given trace. They show how demerit figures of as low as 3% can be achieved by simulating the disk behavior in extreme detail. Similar simulation techniques are used in [12, 15, 22].

DiskSim [6], developed by Ganger, Worthington and Patt, is a general-purpose simulator that goes beyond the techniques of Ruemmler and Wilkes. The DiskSim software requires a large set of parameters to characterize a disk drive. These parameters include nearly a hundred behavioral details and overhead timings, in addition to detailed disk geometry information and a large table of seek times. Much of this complexity stems from DiskSim's goal of simulating the widest possible range of disks. The source code for DiskSim is publicly available, and it has been used as the basis for Dempsey's implementation.

Several recent projects have attempted to automate the process of extracting these large number of disk performance parameters. Many of these rely on a combination of two kinds of techniques [24], namely interrogative extraction and empirical extraction. *Interrogative extraction* makes use of low-level commands in the disk interface to extract information about geometry and other static properties of the disk. Such techniques have successfully been used with many SCSI disk drives [20, 24]. Interrogative extraction, however, is not sufficient for several reasons. First, a given disk drive may not support all interrogative commands. Second, the information returned by such commands may be inaccurate, in which case it is only usable as a hint. *Empirical extraction* observes the behavior of a given disk on a set of carefully-chosen synthetic workloads, and extracts the parameter values from these observations. Since empirical extraction does not depend on specific commands in the disk interface, it is more generally applicable [2, 21].

Dempsey mainly uses empirical extraction techniques to extract the relevant performance and power parameters from a given disk. This makes Dempsey relatively general-purpose and suitable for disks with the IDE interface. The extraction techniques for the performance parameters are quite similar to those described in existing work, like DIXTrac [20]. To these, we add techniques for extracting power parameters.

2.2 Disk Power Management

Most research in disk power management has focused on the behavior of the disk during periods of inactivity, i.e., idle periods. Specifically, the question is when the disk should be put to sleep to minimize power consumption with little impact on performance. Many papers have analyzed the impact of aggressively spinning down disks when the time since last I/O request exceeds some threshold [4, 14, 23]. Algorithms for dynamically varying the spin-down threshold in response to changing user behavior and priorities have also been proposed [3, 7, 10, 13]. IBM's storage systems division has developed an adaptive power management algorithm [1] called ABLE (Adaptive Battery Life Extender). ABLE has been incorporated in IBM 2.5-inch Travelstar drives and IBM Microdrives.

The simulators employed in these projects are simpler. Greenawalt uses an analytical model that assumes that requests arrive according to a Poisson distribution [8]. Helmbold et. al. [10] model power in terms of seconds of activity, rather than using Joules. This simplification relies on two implicit assumptions. One is that a disk has only two distinct power levels: active and idle. The second is that an active disk always consumes power at the same rate. Douglis et. al. [4] use a disk simulator that uses a fixed, average response time for all requests, except for those that lie within a small neighborhood of the previous request. This is quite similar to the model that Reummler and Wilkes show to have a demerit of 35% [19].

These earlier simplified simulations are valuable in studying disk spin-up and spin-down policies. One common assumption shared by these studies is a given disk I/O access pattern generated by a given file system. Researchers, however, have recently begun to investigate how to influence the disk I/O access pattern to reduce energy consumption, sometimes by changing the file system itself. Zheng et. al. [26], for example, analyze the effect of various file system attributes, like data layout policy, burstiness, background data reorganization algorithms, etc., on disk energy consumption. Even when the user issues the same sequence of system calls, the disk I/O requests issued by a log-structured file system, for example, can be very different from those generated by an update-in-place file system, in their locality and burstiness characteristics. A simulator that lacks a level of sophistication that is comparable to that seen in DiskSim, among other disadvantages, may err in its timing estimate of individual I/O requests, which may translate into inaccuracies in its energy estimate. Papathanasiou and Scott [16] explore file-system level techniques for increasing the burstiness of disk accesses. Heath et. al. [9] and Weissel et. al. [23] attempt to achieve similar goals by making applications more "energy-aware". We believe that an accurate and efficient disk power modeling tool like Dempsey can be very useful in projects of this kind. We also expect Dempsey to be useful for systems like ECOSystem [25] where accurate on-line estimates of energy consumption are needed outside a laboratory setting.

3 Dempsey

Dempsey extends the DiskSim simulator with a component to model disk power consumption. The power modeling component is relatively simple, adding fewer than 200 non-comment source lines of code to the existing DiskSim software. Given data files describing the performance and power characteristics of a disk, the simulator can take an input trace file and quickly return an estimate (in Joules) of the energy that would be consumed by executing the given trace on the specified disk. The simulator also produces the standard DiskSim output, which describes performance and response time characteristics.

Dempsey includes tools that automatically extract the required performance and power parameters of a given disk. In all, these tools contain about 2500 source lines of code in C++, Python and bash shell scripts. These tools use the *empirical extraction* technique to extract disk parameters. Specifically, the behavior of the disk on a set of carefully-selected synthetic workloads is observed, and parameter values are extracted from these observations. The performance characterization tools are similar, in nature, to those used in the DIXTrac project [20], so they will be described only in brief in Section 3.2. The power modeling component, however, is described in detail in Section 3.3.

3.1 Measurement Infrastructure

To compute the parameters required for power simulation, the characterization tools need to measure actual power consumption for a variety of synthetic traces. For this, the tools need to interface with a multimeter or a voltmeter. Clearly, it would be preferable to eliminate this need for special equipment, but other observable indicators of power consumption, such as battery life, etc., simply do not provide the speed and accuracy necessary for a detailed understanding of power usage. It is also important, however, to note that the *simulation* component of Dempsey does not require special equipment. Once a disk's power consumption has been characterized, researchers can use the resulting power parameters to model the disk drive without needing to carry out any tests with special equipment. This section describes Dempsey's hardware and software infrastructure.

3.1.1 Hardware

To gather power statistics, we have fashioned a PC card sleeve as pictured in Figure 1, with a schematic diagram in Figure 2. The power measurement sleeve consists of a PC card extender attached to a shunt resistor in series with the card's power supply. Following the general approach taken in [5, 11], we measure the voltage across the shunt resistor. The card extender connects to a PCMCIA-compatible disk drive, such as an IBM Microdrive. The card

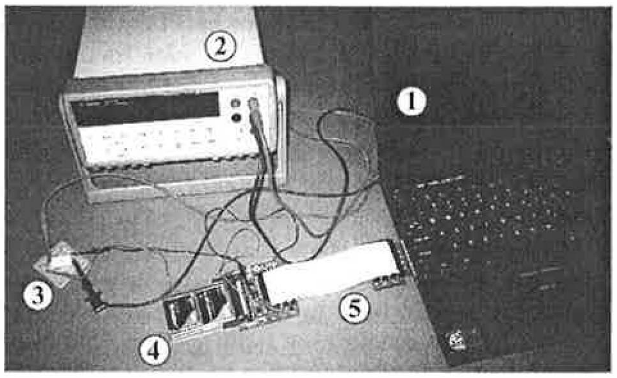


Figure 1: Power measurement apparatus. (1) Linux-based PC with a PCMCIA slot, (2) Digital multimeter, (3) Shunt resistor, (4) IBM Microdrive housed in a PC card adapter, (5) PC card extender.

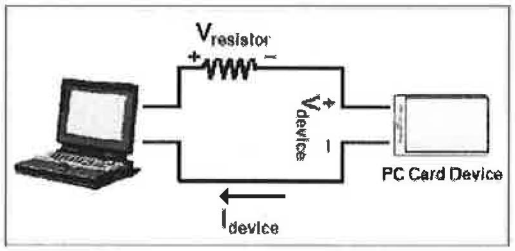


Figure 2: Schematic diagram of the power measurement apparatus.

extender is inserted into a PCMCIA slot of a Linux-based PC. A digital multimeter is used to measure the shunt resistor voltage, and integrate it over a sampling interval to compute the average resistor voltage for that interval. These measurements are logged via a serial link to the PC over the course of an experiment. After independently determining the ohmage of the shunt resistor, we deduce the average current delivered to the disk drive via Ohm's law $I_{Drive} = V_{Resistor}/R_{Resistor}$. This leads to the average power consumed by the drive over the sampling interval using $P_{Drive} = V_{Drive}I_{Drive}$, where V_{Drive} is the voltage of the power supply to the disk drive. The total energy consumed by the drive over the sampling interval is, then, the product of P_{Drive} and the length of the sampling interval.

In all of our measurements, we configure the multimeter to produce 3 samples per second, although the multimeter itself is capable of operating at much higher sampling rates. In other words, the time interval over which average voltage is measured across the shunt resistor is approximately 333 ms. This is a choice in favor of overall accuracy. The multimeter, in producing each sample, needs to spend some time performing computation, and the voltage change during the computation time is basically ignored. So, fast sampling rates, though good for showing peak values, are generally bad for overall accuracy. All of our experiments run several seconds or more. Thus, a relatively long sampling time interval is acceptable in this context.

3.1.2 Software

A typical power measurement experiment has two main tasks: (1) to execute the disk trace, and (2) to record average voltage measurements from the serial port. In the Dempsey setup, these two tasks are performed by two concurrent threads. It is important to ensure that these threads are scheduled fairly during the measurement. To minimize inaccuracy due to unfairness in relative scheduling of the threads, Dempsey schedules both threads as real-time processes, with the port reader having a higher priority than the execution thread. Additionally, the experiments for this paper are run on a Linux system that uses a kernel patched for low-latency and pre-emptible operation. An alternative would have been to use two separate computers, one to execute the trace and the other to record the measurements. The current setup, however, gives sufficient accuracy that we do not use a second computer.

To transfer data to the drive, the execution thread uses the Linux raw device interface (*/dev/raw*), which bypasses the operating system's buffer cache and allows the software to read from or write to arbitrary sectors. The disk also has an internal cache that can interfere with attempts to measure performance. The Microdrive, for example, has both a read cache and a write cache, and it allows a user to disable the write cache, but not the read cache. Unless otherwise noted, the write cache is enabled during the experiments in this paper.

3.2 Performance Modeling

Dempsey uses the DiskSim software to model the execution of a given trace on a given disk. DiskSim models the execution in extreme detail, including modeling different stages of the execution, namely, *seeking*, *rotation*, *data transfer* and *idle periods*. Dempsey adds code to model energy consumption during each of these stages.

To simulate the execution, DiskSim requires specific values for a large number of parameters that characterize disk geometry and layout, mechanical timings and cache behavior, among other things. Dempsey includes performance characterization tools to automatically extract these parameters from a given disk.

Typically, the first step in such characterization is the extraction of detailed information about disk geometry and physical layout of data blocks on the disk. For this, many previous efforts, including DIXTrac [20], have relied on low-level commands in the disk interface that reveal the mapping from logical block addresses (LBAs) to physical locations on the disk. The SCSI interface, for example, has the "Translate" option in its "SEND DIAGNOSTIC" and "RECEIVE DIAGNOSTIC" commands to translate a given LBA to the corresponding physical location.

Dempsey chooses not to rely on such translation commands, since not all disks support such commands. Also, even if a disk does support such commands, the information returned may not be entirely correct and reliable. For example, the IBM Microdrive, which supports the IDE interface, does include a "Translate Sector" command to translate a given LBA to the corresponding cylinder/head/sector (CHS) combination. Unfortunately, this CHS address does not represent the true physical location of the corresponding sector on the disk. Dempsey uses empirical extraction techniques to extract the geometry information, which is then used in the extraction of all other relevant parameters through empirical extraction techniques similar to those used in [20].

3.2.1 Extracting Disk Layout

The most important feature of the mapping between LBAs and their corresponding physical locations on a given disk drive is the drive's *zoning* strategy. Zoning refers to the technique of dividing a drive's surface into several groups (or zones) of tracks, such that all tracks within a given zone have an equal number of sectors. Tracks in different zones may have different numbers of sectors. This allows more sectors on the outermost tracks, which are longer than the innermost tracks. Because LBA to physical-location mappings are generally sequential, an accurate description of a disk's zones is nearly equivalent to a description of a drive's layout, provided that the drive's geometry has not been substantially affected by defects and that the number of heads is known.

Finding tracks. A fundamental building block of the zone discovery algorithm is the `Same_Track`(L) function. `Same_Track` returns a range of LBAs that includes all LBAs on the same track as LBA L. Let `Seek_Time`(L_1, L_2) denote the time taken for a seek from LBA L_1 to LBA L_2. (A function to compute `Seek_Time` can be implemented using the SEEK command directly, so that it bypasses issues of caching and rotational latency.) The `Same_Track` function begins by computing `Seek_Time`(L, L). This represents the time necessary to perform a zero-distance seek, essentially equivalent to the bus and command processing overhead. Any seek of non-zero distance will take substantially longer. The `Same_Track` function can then compute `Seek_Time`(L, L_k) for a series of L_k values. If the `Seek_Time` value is more than twice of the minimal seek time (this factor of two is an arbitrary threshold that proves to work well in practice), then it is inferred that L_k does not lie on the same track as L. `Same_Track` can then use binary search to discover the upper and lower boundaries of the desired track.

From tracks to zones. Note that using the `Same_Track` function, it is trivial to determine the number of sectors on the track on which a given LBA lies. Let `Num_Sectors`(L) denote the number of sectors on the track on which L lies. Remember that a zone is defined to be a set of consecutive tracks, all of which have the same number of sectors. Let `Same_Zone`(L) be a function that returns the range of LBAs that lie in the same zone as LBA L. `Same_Zone` can easily be implemented using the `Num_Sectors` function and binary search.

Starting with the disk's first LBA and repeatedly applying `Same_Zone` function until the disk's final LBA is reached, Dempsey is able to generate a complete map of the disk's zones.

Disk heads. The boundaries, which record the number of blocks per track and the starting and ending LBAs of each zone, are still not enough information to map an LBA to a physical cylinder/head/sector address. On a disk with multiple heads, each cylinder logically contains tracks spread across several different data surfaces. Therefore, if a zone has S_z total sectors and S_t sectors per track, the zone contains $\frac{S_z}{(S_t \times H)}$ cylinders, where H is the number of heads on the disk. Clearly, then, Dempsey must know the number of physical heads on the disk in order to understand its geometry.

The current version of Dempsey relies on the manufacturer's specifications to determine the number of disk heads in a drive, although techniques for experimentally determining this number are also known [20].

3.2.2 Extracting Other Parameters

Other performance parameters, like the seek curve, the rotation speed and the cache parameters, are extracted using techniques similar to those used in the DIXTrac project. Therefore, these techniques are not described here in any detail.

As noted in [20], some disk drives that incorporate *adaptive caching techniques* (where the organization of the on-disk cache changes in response to changing disk access patterns) are not very accurately modeled by DiskSim. Both the IBM Microdrive and the Toshiba HDD are observed to have an adaptive cache. Thus, following [20], Dempsey uses average values for many cache parameters in the parameter files.

3.3 Power Modeling

Traditionally, most approaches to power modeling of hard disks have been very *coarse-grained*. For example, many previous attempts have modeled disk to be in one of two states at any given time: *active* and *sleep*, and power consumption in each of these states has been assumed to be constant. On the other hand, Dempsey's approach is fairly *fine-grained*. Dempsey attempts to accurately estimate the energy consumed by each of several sub-components of a disk request. Each disk request consists of several stages: a seek to the correct cylinder, a period of wait until the disk rotates to the correct position, an actual data transfer, and, possibly, a period of idleness before the next disk request is handled. In this section, we describe how Dempsey computes the values of various power parameters required to model the energy consumption for each of these stages, namely, (1) seeking, (2) rotation, (3) reading, (4) writing, and (5) idle-periods. Table 1 summarizes the power parameters for the two drives. We also describe how Dempsey uses these parameters to arrive at its estimates for the energy consumption for these stages during the execution of a disk trace.

3.3.1 General Approach

To compute the power parameters, Dempsey needs to measure the average power consumption for each of the above-mentioned disk stages. This is a significant challenge because any stage may complete in as little as half a millisecond, whereas the multimeter-based apparatus can, at best, take 75 samples per second. Thus, it is not possible to measure accurately the power consumption of a single stage in a single request, even if the software could somehow identify exactly when the disk transitions between different stages (which itself is a hard problem). Therefore, all of Dempsey's power characterization experiments rely on power measurements taken over the course of longer traces.

Stage	IBM Microdrive	Toshiba HDD
Seeking	637	1287
Rotation	594	1122
Reading	627	1185
Writing	756	1430
Stand-by	61	231

Table 1: Average power consumed (mW). The number for the "Seeking" stage is the average power consumed during a seek over one-third of the maximum seek distance.

Energy consumed during a single sampling interval can be computed using the reading from the multimeter, as described in Section 3.1.1. Total energy consumed by a trace is the sum of the energy consumptions of all the sampling intervals. In order to obtain a measure of the average power consumed by a specific disk stage S, Dempsey runs two traces that differ only in the amount of time spent in stage S. The following formula, then, gives the average power consumption for disk stage S.

$$\overline{P_S} = \frac{E_2 - E_1}{T_2 - T_1}$$

where E_i is the total energy consumed by trace i and T_i is the total time taken by trace i. Note that if the two traces differ only in the time they spend in stage S, then the numerator is the extra energy spent in stage S and the denominator is the extra time spent in stage S. We refer to this method of estimating average power consumption of an individual stage as the *Two-Trace method*.

3.3.2 Seeking

Dempsey generates a Seek-Power Profile to model the seeking stage. The profile lists the energy consumed by the disk for seeks of various distances, measured in number of cylinders crossed. Figure 3 presents the measured Seek-Power Profile for the IBM Microdrive.

Measuring seek power is relatively easier than measuring power consumption for other stages. For this purpose, Dempsey uses the SEEK command directly. To estimate the energy consumed by a seek of C cylinders, the Two-Trace method is used as

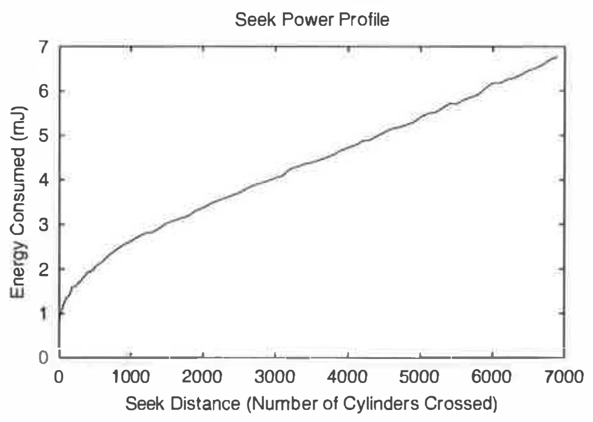


Figure 3: Seek-Power Profile for the IBM Microdrive.

follows. The first trace consists of only SEEK commands addressed to an LBA L chosen randomly at the beginning of the experiment. The second trace alternates evenly between SEEK commands addressed to L and those addressed to another block that is C cylinders away from L. Neither trace spends any time in the rotation, reading, writing or idle periods, so the traces only differ in the seeking stage.

3.3.3 Rotation

The rotation stage of a request occurs after the disk head has reached the correct track. The disk must then wait until its rotation places the desired sector beneath the active head. Although the disk is neither seeking nor transferring data during this interval, it is also not "idle" in the strict sense. During a true idle period, in which no requests are active, a disk may be able to shut down certain components in order to reduce power consumption. Thus, a disk may consume more power during a period of rotation than it does during a true idle period. For that reason, Dempsey includes tests that measure the power consumed while the disk waits for rotation to complete.

For this stage, Dempsey uses a single power parameter, which is a measure of average power consumption during rotation. Again, the Two-Trace method is used. A track T is selected randomly. Both traces issue the same number of single-block writes to T. The write-cache of the disk is disabled during this experiment. The first trace writes blocks on T sequentially, wrapping around when the last block on the track is reached. The second trace writes blocks on T with a larger *stride* between each pair of successive requests.

Each trace contains the same number of single-block write requests, ensuring that they spend equal amounts of time in the writing stage. Neither trace includes any seeking, reading or idle period. Thus, they differ only in their time spent in the rotation stage. Therefore, the Two-Trace method yields the correct power consumption for the rotation stage.

While the exact size of the large stride for the second trace is not particularly important, the stride must be large enough to ensure a substantial difference between the two traces. Dempsey uses a 50-block stride as an arbitrary choice.

3.3.4 Reading and Writing

Dempsey uses one power parameter each for the reading and writing stages. These parameters are a measure of average power consumption during these stages. Dempsey supports two separate methods to compute read and write power consumption. The decision of which method to use depends on whether the disk supports *zero-latency access* or not. When a disk *without* zero-latency access receives a large read or write request, it must wait for the disk to spin until it reaches the *first* sector of the request before it begins transferring data. A disk *with* zero-latency access (also called a read/write-on-arrival disk), on the other hand, can begin transferring data when the disk head is positioned above *any* of the sectors in the request.

To determine whether or not a disk supports zero-latency access, Dempsey issues a single-block read request at the first block of a track, followed by a request to read the entire track, beginning at the first block. In a disk without zero-latency access, this process will take two rotations, whereas in the other case it will take only slightly longer than one rotation. (This same technique is used by DIXTrac [20].)

Measuring for disks without zero-latency access. To measure read power in a disk without zero-latency access, Dempsey again uses the Two-Trace method. A random sequence of distinct tracks is selected. The first trace consists of 1-block reads to the first block in each of the selected tracks. The second trace consists of whole-track reads to each of the selected tracks, where each read begins at the first block in the track. It is easy to see that both traces spend *almost* equal times in the seeking and rotation stages. Neither trace includes any idle time. Thus, the Two-Trace method yields a reasonable approximation of the power consumption parameter for the reading stage.

The power parameter for the writing stage is computed in the same manner, with the write cache of the disk disabled.

Measuring for disks with zero-latency access. It is easy to observe that the traces used above can not be used in this case, because here the two traces will differ significantly in the time they spend in the rotation stage.

To make it work for disks with zero-latency access, we need to change the first trace slightly. Here the first trace, instead of issuing 1-block read commands, issues SEEK commands to each of the tracks in the selected sequence. The second trace issues whole-track reads as in the previous case. It is easy to verify that these two traces, when executed on a zero-latency disk, will differ only in the reading stage. Thus, the Two-Trace method can be applied to obtain the average power consumption of the reading stage. Note that the traces that work in this case do not work for the previous case, because they will differ significantly in the time they spend in the rotation stage when executed on a disk without zero-latency access.

3.3.5 Idle Periods

The behavior of a hard disk during idle periods is substantially more complex than during the disk-stages described above. Typically, several power modes are defined where performance is traded-off for savings in power consumption. For example, many disks have four modes of operation: *active*, *idle*, *standby* and *sleep*. The active-mode is the only mode in which the disk can satisfy requests. The active-mode has the highest power consumption, followed by the idle-mode, the standby-mode and the sleep-mode in that order. The intention is to let the disk operate in one of the low power modes when there is no disk activity, but transitioning from one power mode to another usually incurs time and energy overheads. Thus, power management usually has implications for power and performance (response times, in particular).

Modeling a disk's power management scheme can be decomposed into two tasks. First, the energy and performance overheads associated with mode transitions need to be measured. Second, the power simulator requires a model of when the mode transitions occur.

To measure the cost of mode transitions, Dempsey issues a series of traces to generate an *Idle-Period Energy Profile*. Each trace has a large number of I/O operations that are separated by an idle period of constant length. By varying the length of idle periods across traces, Dempsey generates a series of tuples of the form (t, E, T). A tuple (t, E, T) records the fact that an idle period of length t consumes energy E, and introduces delay T in the response time for the following request. The Idle-Period Energy Profile can be used to determine accurately the modes used by the power management scheme, the overheads incurred during mode transitions, and the power consumed by the disk in different modes.

The task of developing a model for when the disk transitions between modes is more complex. Traditionally, disks have used fixed waiting thresholds to transition from higher power to lower power modes. For such disks, the Idle-Period Energy Profile is sufficient to predict when mode transitions would occur. The fixed thresholds for mode transitions can be inferred from the Idle-Period Energy Profile, and Dempsey provides a default mechanism for determining such fixed thresholds.

Newer disks, however, are moving toward more sophisticated mechanisms for managing their operation. For example, the IBM Microdrive employs the Adaptive Battery Life Extender (ABLE) technology, which, in addition to defining many more power modes, *adaptively* manages transitions between those modes. ABLE continuously monitors disk-request pattern and maintains statistics on the recent history of disk requests. Instead of fixed waiting thresholds, ABLE's decisions on when and how to make power-mode transitions are determined by its predictions about the current request-burst, the current level of internal disk-activity (prefetching, write-behind, etc.), the desired performance-level and the energy costs associated with the transitions.

Ideally, one would like to use the exact ABLE algorithms when modeling the IBM Microdrive in Dempsey, but since these algorithms are not available publicly, we use the fixed-threshold model with the following straightforward implementation. The estimate for the energy consumed during a given idle period of length t is computed by looking up the appropriate tuple in the energy profile. If no tuple is found for length t, then interpolation is used to arrive at an estimate. The time overhead is fed back into the performance modeling component of Dempsey so as to keep the response-time estimates accurate.

3.4 Simulation

Dempsey inherits the performance simulation module from DiskSim, which models the disk in extreme detail. To obtain an estimate of total energy consumption for a given trace, the simulator

Component	Description
Laptop	IBM ThinkPad T20
	750 MHz P3, 128 MB RAM
	Linux Operating System
Multimeter	Agilent 34401A
Shunt Resistor	0.47 Ohm
PC Card Extender	Sycard PCCextend

Table 2: Various hardware components of the experimental setup.

	IBM Microdrive	Toshiba HDD
Model	DSCM-11000	MK5002MPL
Capacity (GB)	1	5
Seek Time (ms)		
Track-to-Track	3	3
Avg. Seek	13	15
Max. Seek	20	26
Rotation (RPM)	3600	3990

Table 3: Detailed characteristics of the IBM Microdrive and the Toshiba HDD.

simply computes estimates for the seeking, rotation, reading, writing and idle-period stages, and adds them up. The Seek-Power Profile is used to arrive at the estimate for the seeking stage. For the rotation, reading and writing stages, the corresponding power parameter is multiplied with the estimated time spent in the stage to arrive at the energy estimate for that stage. The energy spent during the idle periods is estimated as described in Section 3.3.5 above.

4 Experimental Validation

This section presents some experimental results that contribute toward validating the Dempsey simulator for the IBM Microdrive and the Toshiba HDD. We also compare Dempsey against several other alternatives which model the disk power consumption in less detail than Dempsey. We use both synthetic and real-world traces to perform the evaluation. Detailed specifications of various hardware components of the experimental setup and the mobile disks are provided in Tables 2 and 3.

4.1 Synthetic Traces

A disk trace is a sequence of disk requests, each of which has four components: request arrival time, starting block address, size of request and type of request (read/write). This suggests four natural dimensions along which a synthetic trace can be characterized.

- **Delay.** This refers to the probability distribution from which delay intervals between successive requests are chosen. We consider 4 different distributions. (1) *Fixed*: delay interval of a fixed size is always chosen. (2) *Standard*: delay interval is chosen uniformly at random in the range 1-80 ms (this is the distribution used in the DIXTrac validation test [20].) (3) *Long*: here a standard delay is chosen with 0.9 probability and a random delay between 1-8 seconds is chosen with 0.1 probability. (4) *Very Long*: here a random delay is chosen between 1-200 ms with 0.98 probability and between 5-20 seconds with 0.02 probability.

- **Access Pattern.** This determines the sequence of addresses accessed in the trace. Three kinds of access patterns are considered here. (1) *Sequential*: the blocks are sequentially accessed on the disk. (2) *Random*: block addresses in the trace are randomly chosen from the range of valid disk addresses. (3) *Cache Test*: this test is used in [20] to test the accuracy of the disk-cache modeling. In this pattern, 20% of the requests are sequential, 30% are local (i.e., within 250 blocks of their predecessor in either direction), and 50% are completely random.

- **Transfer Size.** This parameter specifies the number of blocks to transfer with each request. The traces used here have two kinds of transfer sizes: (1) *Fixed*, and (2) *Standard*: where size of each request is chosen uniformly at random between 16-24 sectors (i.e., 8-12KB.)

- **Request Type.** This refers to the distribution of read and write requests in the trace. Besides purely read and write traces, the evaluation uses randomly-generated *mixed* traces with read-write ratio of 2:1.

For each synthetic trace, it is necessary to specify a value for each of the four parameters described above. We use eight representative synthetic traces in our experiments. Table 4 summarizes the synthetic traces used. Unspecified parameters in the trace descriptions have the default values given in Table 5. Trace I, with 400 ms delay interval between successive requests, tests the simulator's ability to model medium-length idle periods in which the drives do not enter the low-power modes. Trace

Trace	Description
I	Delay: Fixed at 400 ms
II	Transfer Size: Fixed at 1 sector
III	All Parameters: Standard
IV	Delay: Long
V	Delay: Very Long
VI	Access Pattern: Sequential
VII	Access Pattern: Cache Test
VIII	Delay: Fixed at 100 ms, Access Pattern: Sequential, Transfer Size: Fixed at 128 sectors, Request Type: Only writes

Table 4: Summary of the synthetic traces used.

Parameter	Default Value
Delay	Standard (random in 1-80 ms)
Access Pattern	Random
Transfer Size	Standard (random in 8-12KB)
Request Type	Mixed (read-write ratio of 2:1)

Table 5: Default values for the synthetic trace parameters.

II, with single-sector requests, tests the simulator's accuracy for small requests. Trace III uses the standard parameters (i.e., default values in Table 5) from the DIXTrac traces to test the simulator's handling of mid-sized transfers. Traces IV and V include a number of long idle periods when the drives should enter the low-power modes. Traces VI and VII include non-random access patterns. Trace VIII contains large sequential writes, representing workloads that are typical of log-structured file systems [17].

4.2 Real-world Traces

We use a portion of the 1992 Cello Trace (Disk-2) from HP Labs [18] in our study. The trace from June 12^{th}, 1992 is broken into 6 traces of length 4 hours each. These traces are named A-F in the tables below. Since the Microdrive used has 1 GB capacity, all accesses in these traces to blocks after the first Gigabyte are removed. The traces A-F contain about 130,000 disk requests in all, of which about 81,000 are writes. Table 6 gives detailed characteristics of these traces.

4.3 Alternative Power Models

We compare Dempsey against three less-sophisticated power models. Table 7 lists the values used for the parameters in these models.

Trace	Reads	Writes	Total	Size	Rate
A	529	7460	7989	7	54
B	14942	4433	19375	5	207
C	5590	15619	21209	6	258
D	22107	36077	58184	7	390
E	4351	11765	16116	7	284
F	1742	5689	7431	7	289
Total	49261	81043	130304		

Table 6: Characteristics of the real-world traces. Each trace is 4 hours long. Columns 2-4 describe the number of reads, writes and total number of operations respectively. The fifth column gives the average request size (KB). The last column lists the maximum number of requests in a second in each trace. Note, however, that the disks may not be able to handle requests at these rates, and some requests may be delayed.

Parameter	IBM Microdrive	Toshiba HDD
P_{active}	624 mW	1186 mW
$P_{active-idle}$	531 mW	891 mW
P_{sleep}	61 mW	231 mW
T_0	2 s	15 s

Table 7: Parameter values for the three alternative models.

4.3.1 The 2-Parameter Model

The 2-Parameter model is, in fact, a naive model, which makes two simplifying assumptions: (i) the disk consumes energy at a constant rate when it is actively satisfying disk requests, and (ii) when the disk finishes a disk request and finds that there are no more pending requests in the queue, it immediately enters the sleep-mode. The model uses the following formula to compute an estimate of total energy consumption for a given trace.

$$E_{total} = E_{active} + E_{idle} \tag{1}$$

E_{active} is estimated as $P_{active}T_{active}$, where P_{active} is the assumed power consumption when the disk is active (regardless of whether it is in the seeking, rotation, reading or writing stage), and T_{active} is the time spent by the disk while actively satisfying disk requests. E_{idle} is estimated as $P_{sleep}T_{idle}$, where P_{sleep} is the assumed power consumption in the sleep-mode, and T_{idle} is the length of the entire idle-period in the trace. P_{active} is derived by executing a random trace of 12 KB accesses (evenly mixed with reads and writes), and computing the average power consumed by the trace. P_{sleep} is measured by observing the disk in the sleep-mode after it has been left idle for a long time.

4.3.2 The 3-Parameter Model

The 3-Parameter model improves upon the 2-Parameter model by further refining the modeling of idle periods. As in the 2-Parameter model, equation (1) is used to estimate the total energy consumption, and E_{active} is estimated as $P_{active}T_{active}$. The behavior during the idle periods, however, is modeled as follows. The model assumes that the disk enters an intermediate power-mode, called the "active-idle" mode, as soon as it finishes a disk request and finds no more pending requests in the queue. The transition from the active-idle-mode to the sleep-mode, however, is governed by a fixed waiting threshold T_0. For an idle-period of length L with $L \leq T_0$, the energy consumption is estimated as $P_{active-idle}L$. For an idle-period of length L with $L > T_0$, the energy consumption is estimated as $P_{active-idle}T_0 + P_{sleep}(L - T_0)$. Note that the time or energy overheads of the mode transitions are not modeled. $P_{active-idle}$ is measured by observing the disk immediately after a disk request has finished.

4.3.3 The Coarse-Dempsey Model

The Coarse-Dempsey model is a hybrid between the 3-Parameter model and Dempsey. As before, it uses equation (1) to estimate total energy consumption. E_{active} is estimated as in the 3-Parameter model, whereas E_{idle} is estimated as in Dempsey. Therefore, this model uses the complete Idle-Period Energy Profile of the disk, which also models the time and energy overheads of mode transitions.

4.4 Experimental Results

Tables 8 and 9 present results from executing the synthetic and the real-world traces on the IBM Microdrive. The tables compare the estimates from the power models with the actual (measured) power consumption on these traces. The corresponding results for the Toshiba HDD are presented in Tables 10 and 11.

The foremost conclusion that can be drawn from these results is that Dempsey is able to model disk power consumption quite accurately. For the IBM Microdrive, Dempsey's worst observed error is an underestimate by 7.5%, while the mean error is only 1.8%. For the Toshiba HDD, the corresponding errors are 6.9% and 3.6% respectively.

The simple 2-Parameter model is grossly inadequate. It consistently underestimates the energy consumption because it wrongly assumes that the disk spends all of its non-active time in the sleep-mode with the lowest power consumption, and because it does not model the energy overheads of mode transitions.

For the Microdrive, the 3-Parameter model appears to achieve a vast improvement over the 2-Parameter model in many cases. Here, the worst observed error for the 3-Parameter model is only 10.0%. This highlights the importance of being able to accurately model disk behavior during idle periods. The Coarse-Dempsey model and Dempsey consistently achieve higher accuracy only by being able to model idle periods more accurately. Recall that the 3-Parameter model does not model the energy and time overheads associated with power-mode transitions, which the Coarse-Dempsey model and Dempsey do. This explains why estimates from the 3-Parameter model are consistently lower than those from the Coarse-Dempsey model and Dempsey.

For the Toshiba HDD, however, the 3-Parameter model appears to be less accurate. On Traces IV, V and A-F, which include relatively long idle-periods, the 3-Parameter model overestimates the energy consumption by a large amount. This shows that its modeling of disk behavior during idle-periods is not very accurate. The reason for this is that the Toshiba disk has several *intermediate* power-modes between the highest power active-mode and the lowest power sleep-mode. From the intermediate power-modes, the 3-Parameter model selects the one with the highest power consumption to derive the value for $P_{active-idle}$. Therefore, unless an idle period is very short, the 3-Parameter model stays in a mode with relatively high power consumption.

The only difference between the Coarse-Dempsey model and Dempsey is in the modeling of periods of disk activity. The Coarse-Dempsey model assumes that the disk uses energy at a constant rate when it is satisfying disk requests, regardless of whether it is in the seeking, rotation, reading or writing stage. Dempsey, on the other hand, attempts to separately estimate the energy consumed in each of these stages. As the results show, the two models perform almost equally well on all traces except Trace VIII. Trace VIII is a trace containing only large sequential writes interspersed by idle-periods of 100 ms. It is representative of workloads that may occur more frequently in a different file system (such as an LFS [17]) than those traced in Traces A-F. For this trace, Dempsey is observed to be more accurate than the Coarse-Dempsey model by as much as 5%. Additional experiments with a write-only workload that has little idle time show that the Coarse-Dempsey model can be off by amounts indicative of the power variances shown in Table 1.

Table 12 lists the number of transitions into the

Trace	Actual	2-Parameter		3-Parameter		Coarse-Dempsey		Dempsey	
I	220.1	37.2	(−83.1%)	214.0	(−2.8%)	219.0	(−0.5%)	219.1	(−0.5%)
II	24.4	13.6	(−44.1%)	23.2	(−4.7%)	23.5	(−3.6%)	23.4	(−4.0%)
III	24.6	14.5	(−40.9%)	24.0	(−2.3%)	24.3	(−1.2%)	24.4	(−0.8%)
IV	171.2	42.4	(−75.2%)	160.3	(−6.4%)	170.8	(−0.3%)	170.8	(−0.2%)
V	89.7	31.9	(−64.4%)	84.5	(−5.8%)	87.6	(−2.3%)	87.7	(−2.2%)
VI	22.2	5.8	(−73.7%)	21.2	(−4.3%)	21.7	(−2.4%)	21.6	(−2.5%)
VII	23.8	10.8	(−54.8%)	22.3	(−6.1%)	22.7	(−4.7%)	22.7	(−4.8%)
VIII	8.0	2.3	(−71.3%)	7.2	(−10.0%)	7.2	(−10.0%)	7.4	(−7.5%)

Table 8: Measured and estimated total energy consumption (in Joules) for the synthetic traces executed on the IBM Microdrive. The column labeled "Actual" lists the measured energy consumption. The last four columns, respectively, present the energy consumption estimates from the 2-Parameter model, the 3-Parameter model, the Coarse-Dempsey model and Dempsey. The numbers in parentheses in the last four columns are the percentage difference between the estimated and the actual values.

Trace	Actual	2-Parameter		3-Parameter		Coarse-Dempsey		Dempsey	
A	2116.5	963.0	(−54.5%)	2036.9	(−3.8%)	2136.0	(+0.9%)	2137.9	(+1.0%)
B	1897.9	1030.6	(−45.7%)	1840.6	(−3.0%)	1903.0	(+0.3%)	1902.1	(+0.2%)
C	2462.7	1049.6	(−57.4%)	2370.6	(−3.7%)	2479.4	(+0.7%)	2480.9	(+0.7%)
D	3156.0	1255.9	(−60.2%)	3027.7	(−4.1%)	3152.5	(−0.1%)	3157.3	(+0.0%)
E	2114.3	1022.0	(−51.7%)	2031.4	(−3.9%)	2121.0	(+0.3%)	2122.6	(+0.4%)
F	1801.4	958.4	(−46.8%)	1742.5	(−3.3%)	1813.7	(+0.7%)	1814.4	(+0.7%)

Table 9: Measured and estimated total energy consumption (in Joules) for the real-world traces executed on the IBM Microdrive. The numbers listed have the same meaning as in Table 8.

Trace	Dempsey Simulation Time (s)
A	0.9
B	2.7
C	2.3
D	6.8
E	1.9
F	0.8

Table 13: Time (in seconds) taken by Dempsey to simulate the IBM Microdrive on the real-world traces, each of which is a 4-hour long trace.

sleep-mode predicted by Dempsey and the corresponding numbers from an actual execution of the real-world traces. The "Actual" number of transitions for a given trace is obtained by counting the number of instances in the execution where the response time exceeds a certain threshold. The worst observed error is 3.4% for the IBM Microdrive and 5% for the Toshiba HDD. This shows that Dempsey is able to model the drives' behavior during idle-periods with reasonable accuracy.

Simulation time for Dempsey to simulate the IBM Microdrive on the 4-hour traces are listed in Table 13. These times are measured by running Dempsey on a desktop machine with a 2 GHz Pentium processor. The traces take a total of less than 16 seconds to execute, indicating that Dempsey is able to process traces at the rate of more than 8000 disk-requests per second. Dempsey's memory usage is less than 2 MB. Thus, Dempsey has the potential of being used as an efficient and accurate disk-power modeling tool.

5 Conclusion

In this paper, Dempsey is demonstrated to be able to model hard-disk energy consumption quite efficiently and accurately. Dempsey includes tools that can extract the required parameters from a given disk automatically. This makes Dempsey relatively general-purpose.

Dempsey is experimentally validated for two mobile hard disks, namely, the 1 GB IBM Microdrive and the 5 GB Toshiba Type II PC Card HDD. Both synthetic and real-world traces are used for the evaluation. For the IBM Microdrive, Dempsey's worst observed error is an underestimate by 7.5%, while the mean error is only 1.8%. For the Toshiba HDD, the corresponding errors are 6.9% and 3.6% respectively.

Trace	Actual	2-Parameter		3-Parameter		Coarse-Dempsey		Dempsey	
I	368.0	108.6	(−70.5%)	360.1	(−2.1%)	350.5	(−4.8%)	351.1	(−4.6%)
II	40.4	24.2	(−40.1%)	39.6	(−2.0%)	39.9	(−1.2%)	40.3	(−0.2%)
III	41.4	25.6	(−38.2%)	40.5	(−2.2%)	40.8	(−1.4%)	41.3	(−0.2%)
IV	293.5	126.8	(−56.8%)	432.0	(+47.2%)	277.6	(−5.4%)	278.2	(−5.2%)
V	195.0	89.1	(−54.3%)	280.6	(+43.9%)	189.3	(−2.9%)	189.8	(−2.7%)
VI	36.9	14.2	(−61.5%)	35.3	(−4.3%)	35.8	(−3.0%)	35.6	(−3.5%)
VII	39.7	20.1	(−49.4%)	37.4	(−5.8%)	37.7	(−5.0%)	37.9	(−4.5%)
VIII	14.4	7.1	(−50.7%)	12.5	(−13.2%)	12.6	(−12.5%)	13.4	(−6.9%)

Table 10: Measured and estimated total energy consumption (in Joules) for the synthetic traces executed on the Toshiba HDD. The numbers listed have the same meaning as in Table 8.

Trace	Actual	2-Parameter		3-Parameter		Coarse-Dempsey		Dempsey	
A	5808.0	3338.7	(−42.5%)	8614.2	(+48.3%)	5507.9	(−5.2%)	5511.0	(−5.1%)
B	5206.0	3453.1	(−33.7%)	6884.8	(+32.3%)	4992.9	(−4.1%)	4994.6	(−4.1%)
C	6476.9	3556.0	(−45.1%)	9523.5	(+47.0%)	6114.3	(−5.6%)	6115.0	(−5.6%)
D	6992.9	3914.4	(−44.0%)	10059.3	(+43.8%)	6832.6	(−2.3%)	6834.9	(−2.3%)
E	6115.2	3516.5	(−42.5%)	9305.7	(+52.2%)	5934.4	(−3.0%)	5935.8	(−3.0%)
F	5544.0	3392.5	(−38.8%)	8297.2	(+49.6%)	5421.8	(−2.2%)	5421.9	(−2.2%)

Table 11: Measured and estimated total energy consumption (in Joules) for the real-world traces executed on the Toshiba HDD. The numbers listed have the same meaning as in Table 8.

Dempsey uses a fine-grained approach to model the energy consumption of a disk. In particular, Dempsey attempts to accurately estimate the energy consumption of specific disk stages, namely, seeking, rotation, reading, writing and idle-periods. Dempsey is also compared against several other models, which model disk power consumption in less detail than Dempsey. The results show that accurate modeling of disk behavior during idle periods is critical to the accuracy of any power model. Accurate modeling of periods of activity is also shown to be important, although to a smaller extent.

Acknowledgments

We would like to thank our shepherd Erik Riedel and other FAST referees for their excellent suggestions and many pointers to existing related work. We would also like to thank Russ Joseph for putting together the PC Card sleeve that we could use for our measurements.

References

[1] Adaptive power management for mobile hard drives. Tech. rep., Storage Systems Division, IBM Corporation, April 1999. Available at: http://www.almaden.ibm.com/almaden/pbwhitepaper.pdf.

[2] Aboutabl, M., Agrawala, A. K., and Decotignie, J.-D. Temporally determinate disk access: An experimental approach. In *Measurement and Modeling of Computer Systems* (1998), pp. 280–281.

[3] Douglis, F., Krishnan, P., and Bershad, B. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing* (April 1995), pp. 121–137.

[4] Douglis, F., Krishnan, P., and Marsh, B. Thwarting the power-hungry disk. In *Proceedings of the Winter USENIX Conference* (1994), pp. 292–306.

[5] Farkas, K. I., Flinn, J., Back, G., Grunwald, D., and Anderson, J.-A. M. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proc. International Conference on Measurement and Modeling of Computer Systems* (2000), pp. 252–263.

[6] Ganger, G., Worthington, B., and Patt, Y. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, December 1999. Available at: http://www.ece.cmu.edu/ ganger/disksim/.

[7] Golding, R. A., Bosch, P., Staelin, C., Sullivan, T., and Wilkes, J. Idleness is not sloth. In *Proceedings of the Winter USENIX Conference* (1995), pp. 201–212.

Trace	IBM Microdrive		Toshiba HDD	
	Actual	Dempsey	Actual	Dempsey
A	742	745 (+0.4%)	187	189 (+1.1%)
B	424	428 (+0.9%)	130	134 (+3.1%)
C	757	774 (+2.2%)	220	231 (+5.0%)
D	783	810 (+3.4%)	158	163 (+3.2%)
E	655	661 (+0.9%)	330	344 (+4.2%)
F	529	532 (+0.6%)	318	325 (+2.2%)

Table 12: Number of sleeps. The columns labeled "Actual" give the number of sleeps in a real execution of the trace, while the columns labeled "Dempsey" list the number of sleeps estimated by Dempsey. The numbers in parentheses are the percentage difference between the estimated and the actual values.

[8] GREENAWALT, P. Modeling power management for hard disks. In *Proceedings of the Symposium on Modeling and Simulation of Computer Telecommunication Systems* (1994), pp. 62–66.

[9] HEATH, T., PINHEIRO, E., HOM, J., KREMER, U., AND BIANCHINI, R. Application transformations for energy and performance-aware device management. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2002).

[10] HELMBOLD, D. P., LONG, D. D. E., SCONYERS, T. L., AND SHERROD, B. Adaptive disk spin-down for mobile computers. *Mobile Networks and Applications 5*, 4 (2000), 285–297.

[11] JOSEPH, R., AND MARTONOSI, M. Run-time power estimation in high-performance microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design* (Aug. 2001).

[12] KOTZ, D., TOH, S. B., AND RADHAKRISHNAN, S. A detailed simulation model of the HP 97560 disk drive. Tech. Rep. PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[13] KRISHNAN, P., LONG, P. M., AND VITTER, J. S. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. In *Proceedings of the Twelfth International Conference on Machine Learning* (1995), pp. 322–330.

[14] LI, K., KUMPF, R., HORTON, P., AND ANDERSON, T. E. A quantitative analysis of disk drive power management in portable computers. In *Proc. of Winter USENIX Conference* (January 1994), pp. 279–292.

[15] MULLER, K., AND PASQUALE, J. A high performance multi-structured file system design. In *Proceedings of the 13th ACM Symposium on Operating System Principles* (1991), pp. 56–67.

[16] PAPATHANASIOU, A. E., AND SCOTT, M. L. Increasing disk burstiness for energy efficiency. Tech. Rep. 792, University of Rochester, November 2002.

[17] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[18] RUEMMLER, C., AND WILKES, J. UNIX disk access patterns. In *Proc. of the Winter USENIX Conference* (San Diego, CA, Jan. 1993), Usenix Association, pp. 405–420.

[19] RUEMMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (March 1994), 17–29.

[20] SCHINDLER, J., AND GANGER, G. Automated disk drive characterization. Tech. Rep. CMU-CS-99-176, School of Computer Science, Carnegie Mellon University, December 1999.

[21] TALAGALA, N., ARPACI-DUSSEAU, R. H., AND PATTERSON, D. Microbenchmark-based extraction of local and global disk characteristics. Tech. Rep. CSD-99-1063, University of California, Berkeley, 1999.

[22] THEKKATH, C. A., WILKES, J., AND LAZOWSKA, E. D. Techniques for file system simulation. *Software - Practice and Experience 24*, 11 (1994), 981–999.

[23] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (Dec. 2002), pp. 117–129.

[24] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of SCSI disk drive parameters. In *Proceedings of the ACM Sigmetrics Conference* (1995), pp. 146–156.

[25] ZENG, H., FAN, X., ELLIS, C., LEBECK, A., AND VAHDAT, A. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 2002).

[26] ZHENG, F., GARG, N., SOBTI, S., ZHANG, C., JOSEPH, R., KRISHNAMURTHY, A., AND WANG, R. Considering the energy consumption of mobile storage alternatives. *Submitted for publication, 2002*.

Storage over IP: When Does Hardware Support help?

Prasenjit Sarkar, Sandeep Uttamchandani, Kaladhar Voruganti
IBM Almaden Research Center
San Jose, California, USA
{prsarkar, sandeepu, kaladhar}@us.ibm.com

Abstract

This paper explores the effect of the current generation of hardware support for IP storage area networks on application performance. In this regard, this paper presents a comprehensive analysis of three competing approaches to build an IP storage area network that differ in their level of hardware support: software, TOE (TCP Offload Engine) and HBA (Host Bus Adapter). The software approach is based on the unmodified TCP/IP stacks that are part of a standard operating system distribution. For the two hardware-based approaches (TOE, HBA), we experimented with a range of adapters and chose a representative adapter for the current generation of each of the hardware approaches.

The micro-benchmark analysis reveals that while hardware support does reduce the CPU utilization for large block sizes, the hardware support can itself be a performance bottleneck that hurts throughput and latency with small block sizes. Furthermore, the macro-benchmark analysis demonstrates that while the current generation of the hardware approaches may have the potential to provide performance improvements in CPU-intensive applications, overall the analysis does not demonstrate any performance benefits in database, scientific and email benchmarks. The analysis in this paper points out that a disparity in the processing power between the host and the adapter is the primary cause of the performance bottleneck in the current generation of the hardware approaches. The paper aims to guide the designers of the next generation of hardware-assisted adapters to better leverage the increasing processing power in the host. In particular, future adapters should be capable of handling small operations at wire speed.

1. Introduction

In the past, a typical server-class installation assumed the presence of storage attached to every host system. This type of host system-attached storage relied primarily on the block-based SCSI protocol. The preferred transport for the SCSI protocol in this model was Parallel SCSI where the storage devices were connected to the host system via a cable-based parallel bus. However, as the need for storage grew, the limitations of this technology became obvious. The physical characteristics of the cable limited the number of storage devices as well as the distance of the storage devices from the host system. Also, storage had to be managed on a per-host system basis.

The lack of scalability and manageability of the host system-attached storage model led to the evolution of the concept of a storage area network. In this new model, storage devices are independent entities that provide block storage service via a network to a multitude of host systems. The advent of gigabit networking coupled with the development of high-speed transport protocols further facilitates the service of storage over networks. Most storage area networks use Fibre Channel [Benner96]; other storage area network technologies are Infiniband [Shankley02], VaxClusters [Kronenberg86], HIPPI [Ansi90] and IP [Satran02, Rajagopal02]. A comparison of the principal storage area networking technologies can be found in [Voruganti01].

This paper focuses on storage area networks based on IP networking technology. The advantages of using IP networks are many. First, using the same IP technology for both regular (non-storage) networks as well as storage networks removes the need to have two different types of networks in any infrastructure. Also, the use of a single popular networking infrastructure can leverage widely available network management skills. Second, the presence of well tested and established protocols allow IP networks both wide-area connectivity, scalable routing as well as proven bandwidth sharing capabilities. Third, the emergence of Gigabit Ethernet seems to indicate that the bandwidth requirements of serving storage over a network should not be an issue. Finally, the availability of commodity IP networking infrastructure indicates the cost of building a storage area network will not be prohibitive.

The aim of this paper is to explore the effect of the current generation of hardware support for IP storage area networks on application performance. The paper begins by describing the various approaches possible in IP storage area networks, and focuses on the three prevalent approaches that differ in the level of hardware support. In the software approach, all TCP/IP and storage transport protocol processing is done on the host system. In addition, the software approach relies on unmodified TCP/IP stacks that are part of the standard operating system distribution. In the TOE (TCP Offload Engine) approach, the TCP/IP protocol processing is offloaded to the network adapter while the storage transport protocol processing is done in the host system. Finally, in the HBA (Host Bus Adapter) approach, the entire storage transport protocol processing is offloaded to the network adapter along with TCP/IP protocol processing.

The key contribution of this paper is to compare these three approaches for IP storage area networks with the help of micro-benchmarks and macro-benchmarks. In the micro-benchmark analysis, the three approaches were compared with respect to latency and throughput by measuring their sensitivity to block sizes as well as CPU, I/O bus and memory speeds. The micro-benchmark analysis was projected onto the real world by running database, scientific and email macro-benchmarks on each of the three approaches. To obtain a representative adapter for each of the hardware approaches, we experimented with a range of adapters and then chose the one with the best performance profile for the micro-benchmark and macro-benchmark analysis.

The results show that contrary to intuition, the representative adapters of the current generation of the hardware approaches are not inherently superior in terms of performance, which is surprising given the cost of hardware offload. The results indicate that while the hardware support decreases the CPU utilization-to-throughput ratio for large block sizes, the hardware support can itself be a performance bottleneck that hurts the rate of I/O operations in comparison to the software approach for small block sizes. This performance bottleneck can be isolated to the disparity in computing power between the host and the current generation of the hardware-assisted adapters. Consequently, the current generation of the hardware approaches is not superior in terms of latency and throughput. This phenomenon is also observed in database, email and scientific benchmarks. This calls for the need for intelligent hardware support that can take advantage of the increased computing power of general-purpose processors.

2. IP Storage

The emerging field of IP storage area networks has the necessary technical infrastructure that makes it possible to transport block storage traffic:

A high-bandwidth scalable network interconnect. A storage area network must provide high network bandwidths for storage to be delivered as a scalable service to applications residing in host systems. In the context of IP networks, Gigabit Ethernet can provide the necessary infrastructure for a high-bandwidth storage area network.

Reliable delivery. A storage area network needs a reliable transport protocol to exchange control and data between the host systems and the storage devices. Fortunately, the IP networking community has invested a lot of research into building a widely-deployed, reliable and in-order transport protocol called TCP. With this in mind, the architects for IP storage area networks chose TCP as the primary transport protocol rather than pursue the time-consuming approach of inventing, deploying and fine-tuning a specialized protocol for storage transport.

Security and management. The IP networking infrastructure has support for security and management protocols that address the needs of a storage area network. SSL, Kerberos and IPSec are some of the available security mechanisms. In terms of management, DNS allows for unique worldwide naming, SLP for discovery of resources on an IP network, and SNMP and SMI for monitoring and diagnosis of IP network nodes.

It should be noted here that IP storage area networks focus on block service in contrast to network file systems that provide remote file access over IP networks.

2.1. Approaches to IP Storage Area Networks

There are three main approaches to building an IP storage area network, each with its distinct performance characteristics.

2.1.1. Software

The *software approach* envisages using a software TCP/IP stack for storage transport. Proponents of this approach claim that performance should scale with ever-increasing CPU speeds, obviating the need for any hardware assistance. However, preliminary results [Sarkar02] using this approach indicate that the main performance bottleneck is the high CPU utilization involved in large block transfers. The two major components of this high CPU utilization are:

- Interrupt overhead due to the high rate of Ethernet frame-sized transfers from the adapter to the host.
- The TCP copy-and-checksum overhead for large block transfers.

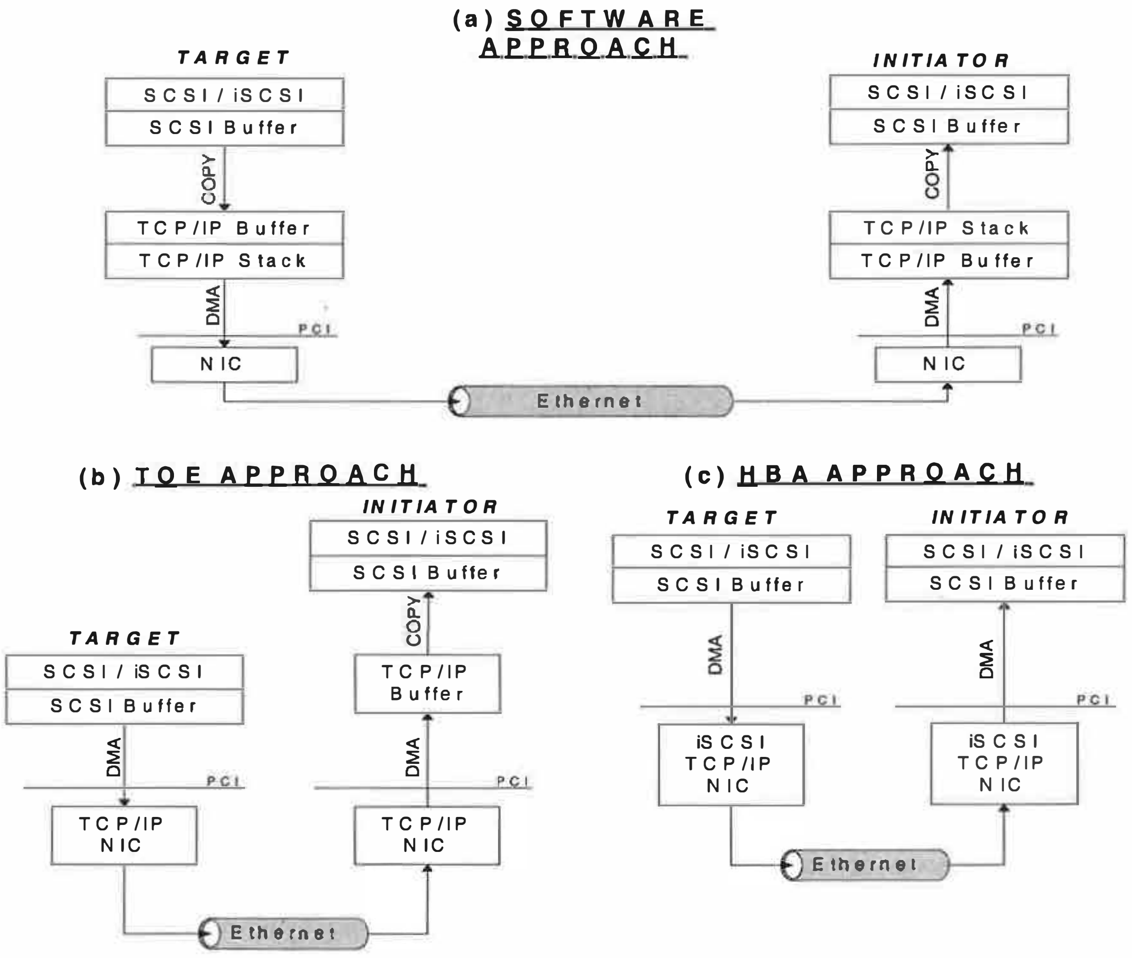


Figure 1. This figure shows the flow of a read response in each of the three approaches to IP storage for both the initiator and target (assuming that a read request has been made earlier). The storage transport protocol in this figure is assumed to be iSCSI. The software approach is shown in part (a) where the SCSI/iSCSI stack in the target copies the requested SCSI buffer into a TCP/IP buffer for transmission by the TCP/IP stack. The TCP/IP buffer is then DMA-ed onto the network adapter card where it is transmitted over the network. The receiving NIC then DMA-s this buffer into a TCP/IP buffer on the host for TCP/IP stack processing, after which the buffer is copied into the destination SCSI buffer for SCSI/iSCSI stack processing. The TOE approach is shown in part (b) where the SCSI/iSCSI stack in the target DMA-s the requested SCSI buffer directly onto a TCP/IP-capable NIC, where it is transmitted after the requisite TCP/IP protocol processing. On receiving the buffer, the TCP/IP-capable NIC on the receiving side DMA-s the buffer onto an anonymous TCP/IP buffer on the target. This anonymous buffer is then copied into the destination SCSI buffer for SCSI/iSCSI stack processing. The HBA approach is shown in part (c) where the SCSI stack in the target DMA's the requested buffer into the iSCSI-capable NIC for iSCSI/TCP/IP processing and transmission over the network. On receiving the buffer, the iSCSI-capable NIC in the receiving side performs iSCSI/TCP/IP protocol processing to learn the identity of the destination buffer and directly DMA-s to this destination buffer for SCSI protocol processing.

The *jumbo frame* approach is a variant of the software approach and improves on the software approach by using 9KB Jumbo Ethernet frames to reduce the per-packet overhead. However, Jumbo Ethernet frames are controversial as detractors claim that large frame sizes are detrimental to efficient routing and quality of service. Due to a lack of consensus, Jumbo Ethernet frames are not standardized and may not be present in the future. Consequently, this approach is not examined further in the paper.

Yet another variant of the software approach is the *zero-copy approach* which uses modified TCP/IP stacks with zero-copy transmit capability. This approach reduces the TCP copy-and-checksum overhead as the responsibility of generating the checksum is offloaded to the network adapter. However, zero-copy receives are typically not possible on such stacks because the network adapters are unaware of the final destination of any frame. Re-mapping the network buffer onto an application buffer can remove the copy on the receive path. However, issues with page-alignment and virtual memory costs (particularly in SMP environments) have hindered adoption in production operating systems. Since the primary overheads on storage area networks occur on the receive path [Sarkar02], this approach is also not examined further in the paper because of the lack of stable support for zero-copy receives.

2.1.2. TOE

The *TOE approach* involves network adapters with TCP/IP offload engines where the entire TCP/IP stack is offloaded onto the network adapter. This also reduces the TCP copy-and-checksum overhead. The interrupt overhead is also reduced because of the adapter generates at most one interrupt for every large block transfer. However, zero-copy receives are usually not possible on such stacks because the TCP/IP stack is also typically unaware of the final destination of any TCP/IP packet, though the discussion of re-mapping the network buffer in Section 2.1.1 is also relevant here. Another complication for zero-copy receives is the presence of higher-level protocol headers in the data stream that complicates buffer alignment.

2.1.3. HBA

The *HBA approach* envisages the use of network adapters that have a specific storage transport interface (such as iSCSI) and is aware of the storage protocol semantics. This approach will also reduce the interrupt overhead, as the network adapter will ensure at most one interrupt per data transfer. More importantly, the protocol-specific direct data placement support in the adapter ensures that there are no copies on the receive path. As with the TOE approach, offloading the protocol processing to the adapter will eliminate the TCP/IP copy-and-checksum overhead.

The HBA approach can be envisaged as a specialized version of the RDMA approach [Bailey02, Compaq97], which provides zero-copy receive support via direct data placement to any transport protocol. However, in terms of performance analysis, the HBA and RDMA approaches both provide zero-copy receives and TCP/IP offload and are functionally similar.

The software, TOE and HBA approaches are also currently the mainstream in IP storage area networks. Figure 1 shows the data flow of a read response in the software, TOE, and HBA approaches to better exemplify the differences between the approaches.

The rest of this paper compares the software, TOE and HBA approaches in terms of both micro-benchmarks and macro-benchmarks. The goal is to identify whether the incremental hardware support necessarily improves performance. The micro-benchmarks do a sensitivity analysis of each of these approaches with respect to block sizes, CPU, memory and I/O bus speeds so as to identify potential performance bottlenecks. The macro-benchmarks project the various performance characteristics of each of these approaches onto real-world applications.

3. Micro-benchmarks

3.1. Experimental Setup

To evaluate the performance of IP storage, the protocol of choice was iSCSI [Satran02]. The iSCSI protocol is an IETF proposed standard for transporting SCSI over TCP/IP. There are sufficient iSCSI products available from many industry vendors to do an experimental analysis of all three approaches. For the micro-benchmark analysis, the experimental setup consisted of an iSCSI initiator workstation connected through an Alteon 180 Gigabit Ethernet switch to an iSCSI target server.

3.1.1. Initiator Workstation Setup

The iSCSI initiator workstation was powered by an AMD Athlon MP 1900 CPU (1.6 GHz CPU clock speed) with 4 GB of PC2100 DDR memory, and supported both 64-bit 66 MHz and 32-bit 33 MHz PCI slots. The Athlon 1900 CPU was in the 48-th percentile of all the available CPUs in terms of SpecInt performance at the time of testing. The motherboard in the initiator workstation allowed the CPU front-side bus (FSB) speed to be varied from 90 MHz to 141 MHz with a fixed CPU clock multiplier; this enabled CPU speed variations from 1.1 GHz to 1.7 GHz. The operating system running on the iSCSI initiator workstation was Windows 2000 SP3. There were no modifications made to the TCP/IP stack or any other kernel component.

The initiator workstation was configured with all the three approaches: software, TOE, and HBA.

The network adapter for each approach was chosen after evaluating five HBA and TOE cards from different manufacturer. All of these products were released on 2002 and represent the state of the art. We performed read-cache hit benchmarks with the block sizes of 512 bytes and 64 KB from the initiator workstation to the target server to get an initial performance profile of each adapter. We used this benchmarks to select the best adapter for each of the hardware approaches. The percentage difference between the best performing adapter and the worst performing adapter on the 512 byte and 64 KB read-cache hit test was 12% and 15% respectively.

In the software approach, the initiator workstation ran the IBM Windows initiator v1.2.2 kernel-mode driver that implemented draft version 8 of the iSCSI standard. The kernel-mode driver did not have support for zero-copy receives because of buffer alignment issues. An Intel Pro/1000F Server NIC provided the connectivity. The checksum offloading feature of the NIC card was not utilized.

In the TOE approach, the selected TOE adapter replaced the Intel NIC card. This TOE card provided ASIC support to offload the fast-path TCP/IP functionality from the Windows TCP/IP stack. With the TOE card, the iSCSI software used was still the IBM Windows initiator v1.2.2 except that the fast-path TCP functionality was taken over by a third-generation ASIC in the TOE card.

In the HBA approach, the selected iSCSI HBA card replaced the TOE card. The TOE engine on the HBA card has an ASIC chip that provides TOE functionality. Firmware running on a CPU on-board the HBA card provides an implementation of the draft version 8 of the iSCSI standard, though data transfers between the host system and the HBA do not involve the CPU.

Unless otherwise mentioned, the adapter cards for the three approaches were placed in a 64-bit 66 MHz PCI slot. In addition, the default CPU speed was always 1.6 GHz (133 MHz FSB speed). The maximum number of outstanding I/Os in the iSCSI protocol was set to 60 in each of the three approaches. Since jumbo frames were not universally supported, we used the default Ethernet frame size of 1.5K.

3.1.2. Target Server Setup

The iSCSI target server was powered by a dual-800 MHz Pentium III CPU configuration with 1 GB of PC133 SDRAM memory, and supported both 64-bit 66 MHz and 32-bit 33 MHz PCI slots. The target server was equipped with an IBM ServeRAID 4H SCSI PCI RAID controller card with 48 36-GB 10,000-RPM SCSI disks. An Intel Pro/1000F NIC provided the Gigabit Ethernet connectivity. Both cards were placed in 64-bit 66 MHz PCI slots in different PCI buses to avoid I/O bus contention. The operating system running on the target server was Linux 2.4.2-2 with no modification to the TCP/IP stack or any other kernel component.

The target server was only configured with the software approach while the approaches were varied in the initiator workstation to better identify the performance variation in the three approaches. Furthermore, the results of the performance analysis in Sections 3 and 4 also validate the fairness in the choice of the software approach in the target server. The target server ran an iSCSI server kernel daemon (IBM target v1.2.2) that implemented draft version 8 of the iSCSI standard. The kernel daemon also provided read caching functionality that allowed repeated read requests for blocks to be satisfied from the host memory of the iSCSI target server rather than from the RAID controller. The target server also provided support for write-back caching.

3.1.3. Measurement Tool

In the micro-benchmark experiments, the Iometer measurement application [Intel02] on the initiator workstation issued read() calls via the unbuffered block interface (ASPI) to the SCSI layer. At this layer, the read() call got translated to the corresponding SCSI commands and was sent to the low-level iSCSI driver. The use of the unbuffered block interface inhibits the use of caching in the Initiator workstation memory. This allows us to better measure the cost of transporting SCSI over TCP/IP without being polluted by cache effects in the initiator workstation. Experiments were also performed using write() calls but as the results did not reveal anything beyond the available conclusions.

In addition, all the read requests were directed to the same disk block address to take advantage of caching at the iSCSI target server. The reason for using cached read requests was to make sure that the results were focused on measuring storage transport efficiency and would not be contaminated by RAID and disk performance issues.

3.2. Performance Analysis

The metrics used to compare the three approaches to IP storage were throughput and latency. The throughput was measured using 16 worker threads in Iometer. A larger number of worker threads were not used because the aggregate throughput did not increase beyond this number. Each thread in the measurement application issued 100,000 sequential read commands to the same disk block address and the aggregate throughput was measured on completion of the reads by all threads.

The latency measurements were performed using 1 worker thread in the Iometer measurement application (Section 3.1). The worker thread in Iometer issued 100,000 sequential read commands to the same disk block address and the latency was measured by dividing the elapsed time by the total number of commands. The CPU utilization at the initiator workstation and target server was also measured.

3.2.1. Block Size Sensitivity

The first experiment in the micro-benchmark performance analysis pertains to the sensitivity of throughput to variations in the block size used by the Iometer measurement tool. The block size was varied from 0.5 KB to 64 KB and the resultant throughput is shown in Figure 2. The corresponding initiator workstation CPU utilization is shown on the right-hand side in the same figure. The target CPU was not saturated in any experiment.

The results show that the software approach achieves the best numbers in terms of throughput, though the initiator CPU is completely saturated for the lower block sizes of 0.5 KB to 8 KB. The question of whether a faster CPU can aid the performance is investigated in Section 3.2.2. In the larger block sizes of 16 KB to 64 KB, the performance of the software approach is limited by a resource threshold which can be attributed to either the PCI bus or the memory as the Intel adapter is capable of higher throughput. This resource threshold is investigated by using the I/O bus and memory speed sensitivity experiments in Sections 3.2.3 and 3.2.4.

The initiator workstation CPU utilization-to-throughput ratio is of particular importance to applications that are sensitive to CPU cycle availability. These applications benefit only when the throughput is high and the CPU utilization-to-throughput ratio is low. Both hardware approaches show lower ratios of initiator workstation CPU utilization-to-throughput, particularly when the block sizes are large. For example, at the 64 KB block size, the ratio for the TOE approach is just 52% of that of the software approach, though at the 4 KB and 0.5 KB block sizes, the ratio for the TOE approach is 77% and 96% of that of the software approach respectively. Similarly, the ratio of the HBA approach for the 64 KB, 4 KB and 0.5 KB block sizes is 17%, 73% and 113% of that of the software approach respectively. When the block size is large, the per-byte costs of the software approach due to TCP/IP copy-and-checksum and interrupt overhead (Section 2.1) increase the CPU utilization-to-throughput ratio. However, when the block size is small, the per-byte costs of the software approach are competitive with that of the hardware approaches resulting in comparable CPU utilization-to-throughput ratios.

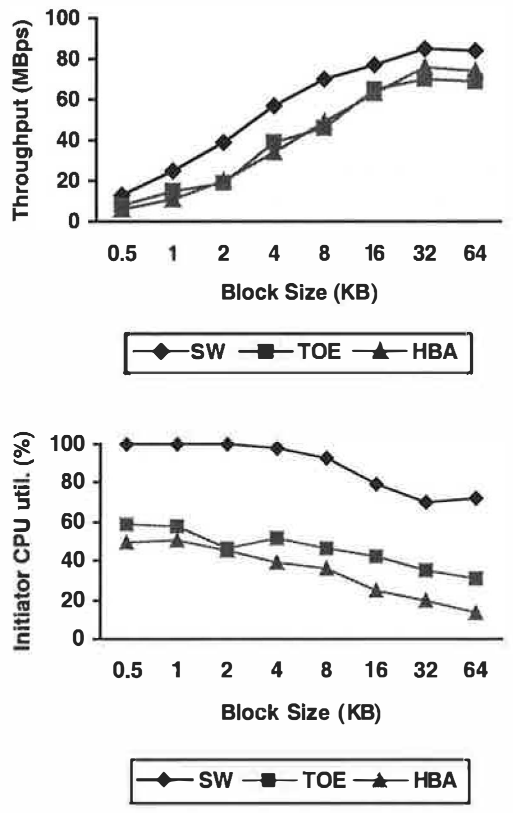


Figure 2. The figure shows the sensitivity of the throughput in each of the three approaches in relation to block size. The resultant initiator workstation CPU utilization is shown on the right.

The throughput of the hardware approaches does not match that of the software approach for any block size. It was observed that neither the initiator workstation CPU nor the target server CPU is completely utilized. Moreover, as the performance differential is also present when the block size is small (0.5 KB) and the throughput is not very high (~10 MBps), the PCI bus and memory speeds can be ruled out as a cause for the inferior performance of the hardware approaches. Consequently, the performance bottleneck can be pinpointed to the hardware offload in the TOE and HBA approaches.

The latency analysis shown in Table 1 reconfirms the performance bottleneck in the hardware offload for the TOE and HBA approaches. As was discussed in Section 2.1, the software approach has a high per-byte cost compared to the hardware approaches due to the TCP/IP copy-and-checksum and the interrupt overhead in the receive path. In the smaller block sizes where the per-operation costs dominate per-byte costs, the software approach is clearly superior in terms of latency. When the block size is large, the per-byte costs dominate the per-operation costs and the superiority of the software approach is narrowed. The latency in the HBA approach is particularly high because of the involvement of the slow StrongARM CPU. However this is not considered inherent to the HBA approach as alternative HBA designs have merged iSCSI and TOE functions into a single ASIC.

	Latency (ms)			
Block Size (KB)	*0.5*	*4*	*8*	*64*
Software	0.12	0.17	0.22	0.97
TOE	0.17	0.26	0.28	1.01
HBA	0.41	0.47	0.51	1.52

Table 1. The table shows the sensitivity of the latency in each of the three approaches in relation to block size

The performance bottleneck in the hardware approaches could be either in the software drivers or in the offloaded protocol processing engines. To further analyze the bottleneck, we measured the per-operation cost in the host systems for each of these approaches. We conducted an experiment on the default setup with a block size of 512 bytes to reduce the per-byte costs to a minimum. We then measured the CPU utilization and the rate of operations for reads on the target server for this particular block size. The number of threads in this read experiment was limited to one so as to remove the effect of interrupt coalescing and get a clearer picture of per-operation costs.

Table 2 shows that the initiator workstation CPU overhead of a single operation for the hardware approaches is less than that of the software approaches. This indicates that the software driver overhead does not contribute to the performance bottleneck in the hardware approaches. Consequently, the high per-operation costs in the hardware approaches can be attributed to a mismatch in processing speeds between the host system and the hardware offload.

	Ops per second	**Initiator CPU util. (%)**	**Initiator CPU util. per op (%)**
Software	8267	38	0.0046
TOE	5959	22	0.0036
HBA	2580	7	0.0027

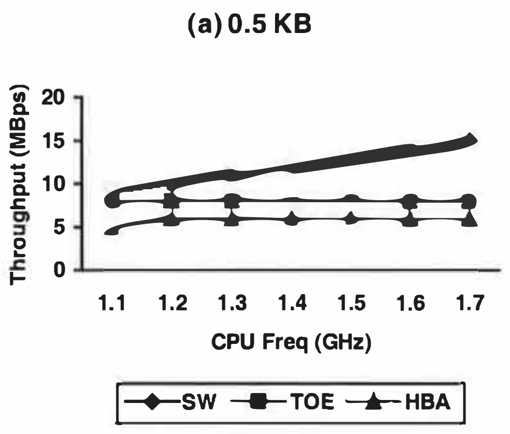
Table 2. The table shows the sensitivity of the rate of operations, the initiator workstation CPU utilization and the per-operation initiator workstation CPU utilization for a single-threaded 0.5 KB read test for each of the three approaches to IP storage.

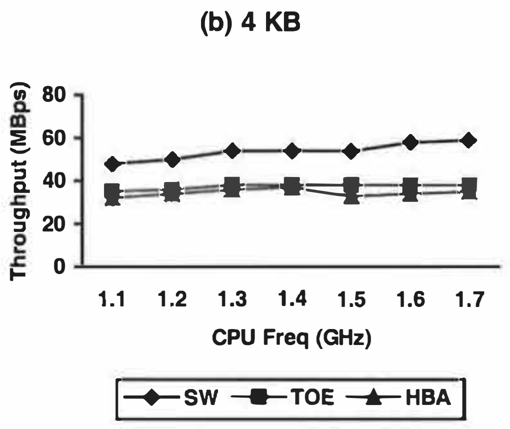
3.2.2. CPU speed sensitivity

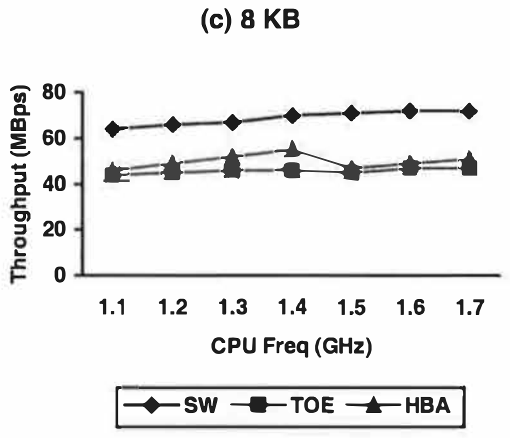
The second experiment in the micro-benchmark performance analysis measured the sensitivity of throughput to the CPU speed in the initiator workstation. In this experiment, the CPU FSB speed was varied from 92 MHz to 141 MHz with a fixed clock multiplier, resulting in an effective CPU speed variation from 1.1 GHz to 1.7 GHz. The throughput was measured with the block sizes of 0.5 KB, 4 KB, 8 KB and 64 KB for each of the three approaches.

The results show that performance of the software approach scales with increasing CPU speeds, particularly for the smaller block sizes. However, when the block size is 64 KB, the resource bottleneck (investigated in the following sub-sections) prevents any scaling of throughput with respect to CPU speeds. The performance of the TOE approach is marginally sensitive to increasing CPU speeds while that of the HBA approach shows no sensitivity at all. This is to be expected due to the offload of protocol processing onto the adapter cards in the hardware approaches.

A similar effect was also observed in the sensitivity of the latency of the three approaches to increasing CPU speeds.







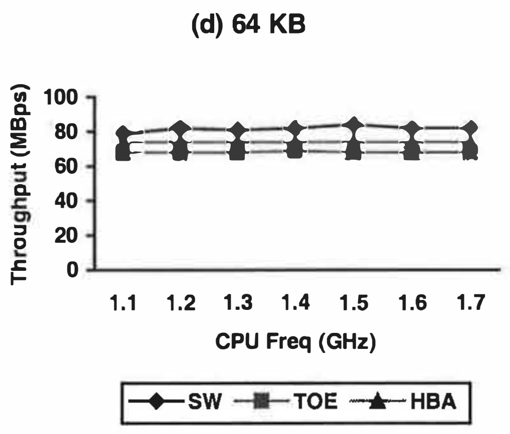


Figure 3. This figure shows the sensitivity of the throughput in relation to CPU speeds (denoted in terms of CPU frequency) in each of the three approaches for the block sizes of (a) 0.5 KB, (b) 4 KB, (c) 8 KB, and (d) 64 KB.

3.2.3. I/O bus speed sensitivity

The third experiment in the micro-benchmark performance analysis measured the sensitivity of throughput to the PCI bus speed in the initiator workstation. The relevant adapter in each of the three approaches was placed alternately in a 32-bit 33 MHz PCI bus and a 64-bit 66 MHz PCI bus. The throughput was measured with the block sizes of 0.5 KB, 4 KB, 8 KB and 64 KB to observe the PCI bus effect.

The results are shown in Figure 4 and indicate that all the approaches are sensitive to the PCI bus speed of the slot holding the adapter, particularly at the 64 KB block size. However, the software approach is the most sensitive to the PCI bus speed as the throughput drops 28% for the 64 KB block size when the PCI bus speed is lowered from 64-bit 66 MHz to 32-bit 33 MHz. The corresponding numbers for the TOE and HBA approach are 18% and 10% respectively. This is due to the fact that the PCI overhead is amortized over 64 KB transfers in the hardware approaches, while the software approach incurs the same overhead over Ethernet frame-sized transfers (1.5 KB). The impact of the difference in overhead is more visible in 32-bit 33 MHz PCI because of the slower PCI bus speed.

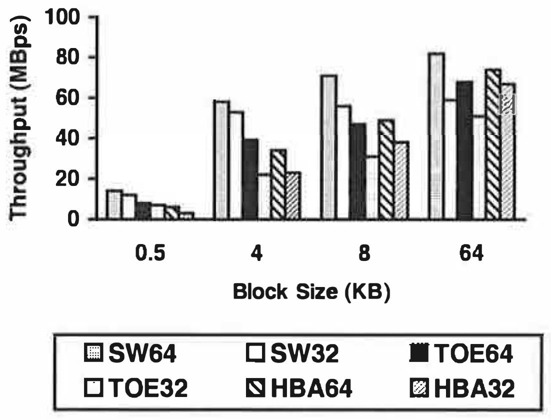


Figure 4. This figure shows the sensitivity of the throughput analysis with respect to PCI bus speeds. The legends SW64, TOE64 and HBA64 represent the throughput in relation to block size when the corresponding adapter is placed in a 64-bit 66 MHz PCI bus slot. Similarly, the legends SW32, TOE32 and HBA32 represent the throughput in relation to block size when the corresponding adapter is placed in a 32-bit 33 MHz PCI bus slot. The legends from left to right are: SW64, SW32, TOE64, TOE32, HBA64, HBA32.

The software approach shows no improvement in throughput when the CPU speed is varied with the adapter at the 32-bit 33 MHz PCI bus slot. A similar phenomenon was observed in Section 3.2.2 when the adapter was placed in a 64-bit 66 MHz PCI slot. This indicates that the PCI bus speed is an important factor in the resource threshold described in Section 3.2.1.

The latency analysis of the three approaches with regards to the sensitivity to the PCI bus was also measured. The analysis showed that the latency differential between the two PCI buses is marginal (< 5%) for all approaches when the block size used is small (0.5 KB, 4KB) but increases to as much as 50% when the block size is large (64 KB).

This experiment shows that the PCI bus speed is an important performance factor in overall performance, particularly the software approach.

3.2.4. Memory speed sensitivity

The final experiment in the micro-benchmark performance analysis measured the sensitivity of throughput and latency to the memory speed in the initiator workstation. The latency and throughput measurements were taken with the default DDR2100 memory (266 MHz) as well as the alternate DDR1600 memory (200 MHz) in the initiator workstation. The throughput and latency was measured with the block sizes of 0.5 KB, 4 KB, 8 KB and 64 KB to observe the memory speed effect. No statistically significant differences were found between the results of the throughput and latency analysis for both the memory speeds, indicating that the current memory speeds are not a resource bottleneck at these rates of throughput. This further indicates that the principal resource threshold discussed in Section 3.2.1 is the PCI bus speed.

4. Macro-benchmarks

4.1. Experimental Setup

The macro-benchmark analysis attempts to project the results obtained from Section 3 onto application performance. For the macro-benchmark analysis, the experimental setup was identical to that of the micro-benchmark analysis. As in Section 3, the target server was only configured with the software approach while the approaches were varied in the initiator workstation.

4.2. Performance Analysis

The TPC-C, Postmark and TPIE-Merge benchmarks were used to evaluate the software, TOE and HBA approaches so as to capture the primary application categories of databases and email.

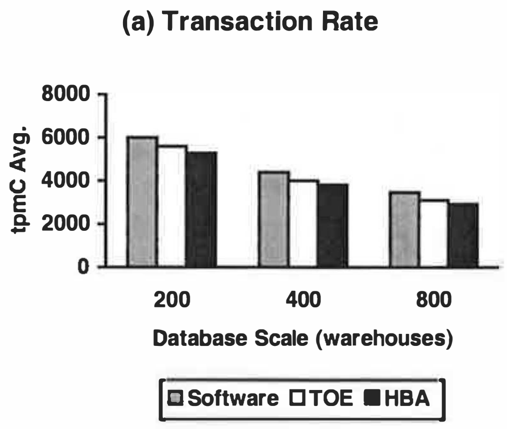
4.2.1. TPC-C

The TPC Benchmark C or TPC-C [TPC97] is an online transaction processing (OLTP) benchmark. Multiple transaction types, database complexity, and overall execution structure distinguish the TPC-C benchmark. The metric for measuring performance is number of transactions completed per minute (tpmC). The benchmark specifies a method for scaling the database based on an assumed business expansion path of the supplier.

The database used was DB2 v7.2 Enterprise Edition. The database settings were fine-tuned for performance on the initiator workstation configuration based on suggestions from IBM Toronto Laboratory staff. For this analysis, we varied the number of database warehouses from 200 to 800. The database resides in a single filesystem spanning 42 drives on the target server. An experimental run of the benchmark with the 42 drives attached as local disks completely saturated the initiator workstation CPU. This indicates that the benchmark is well tuned and not limited by disk drive performance in this particular configuration. The TPC-C benchmark is characterized by highly multi-threaded random page-sized (4 KB) I/Os with reads dominating writes.

The results (Figure 5(a)) show that the software approach has better results compared to the hardware approaches even for this CPU-intensive benchmark. Even though the software approach has a higher CPU utilization-to-throughput ratio for the page-sized transfers used in the benchmark, the software approach is able to compensate for this by a higher rate of I/O operations at this block size as seen in Section 3.2.1. The effective bandwidth of the three approaches is directly proportional to the transaction rate in the benchmark (tpmC).

Figure 5(b) shows that the software approach has a higher average CPU utilization during the benchmark. The average CPU utilization of all the approaches in this particular benchmark does not saturate the initiator CPU because of a limit on the maximum concurrency in the iSCSI protocol (60). Since there was no means to increase the limit, it was not possible to measure the relative performance with a saturated initiator CPU.



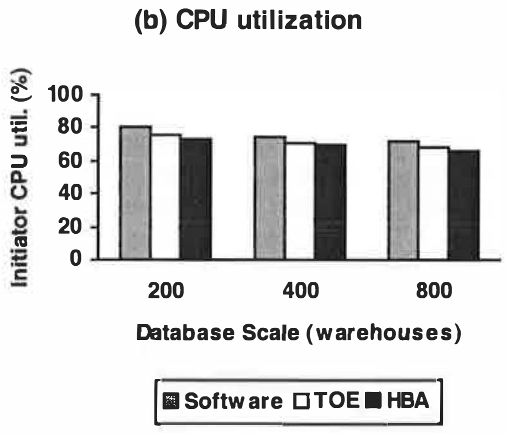


Figure 5. (a) This figure shows the average transactions completed per minute for the three approaches: Software, TOE and HBA when running the TPC-C benchmark for a database scale that spans 200 to 800 warehouses. (b) This figure shows the average CPU utilization in the initiator workstation for the experiments described in (a).

4.2.2 PostMark

The PostMark benchmark [Katcher97] simulates workloads that capture the storage behavior of electronic mail, news and web commerce applications. The benchmark consists of creating files, performing read and write operations (called transactions), and then deleting the files. The benchmark allows for the specification of the number of files to be created, file size, read and write probability, number of transactions, and unit of data transfer between the client and the server. However as the benchmark does not have any application processing, it cannot be considered a true email, news or web commerce application. In contrast to the previous benchmark, this benchmark is single-threaded.

The PostMark benchmark was run with the default parameters except that the number of files was reduced to 150 so as to remove the effect of file system lookup efficiency from storage transport protocol analysis. As the file sizes were varied, there was not much difference between the performances of the three approaches for the small file sizes (<= 64 KB). This was not unexpected as the overhead of setting up a PostMark transaction was comparable to the actual PostMark transaction cost. However, for larger file sizes (> 64 KB), the software approach starts showing superiority in terms of time to run the benchmark by as much as 40% over the hardware approaches (shown in Figure 6) because of the superior latency in the block sizes used by the benchmark.

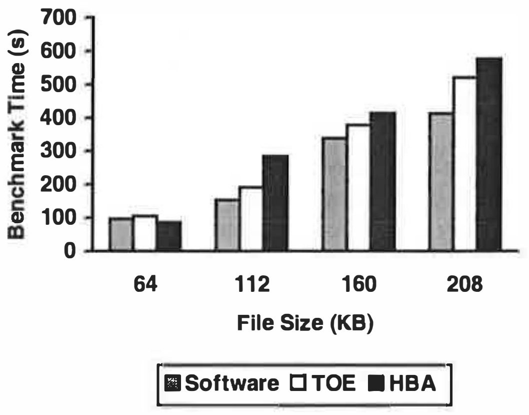


Figure 6. This figure shows the time to complete the PostMark benchmark for the three approaches: Software, TOE and HBA when the size for 150 files is varied from 64 KB to 208 KB in increments of 48 KB. The remaining PostMark parameters are the default.

4.2.3 TPIE-Merge

The TPIE benchmark [Vengroff96] combines raw sequential read/write I/O operations with application processing. The objective behind using this benchmark is to assess the I/O performance of the different approaches when the application processing has high CPU utilization. This benchmark will penalize any approach that has a high CPU utilization-to-throughput ratio and favors efficient storage transport protocols.

The TPIE toolkit provides scan, merge and sort routines that can be used to generate I/O and application processing scenarios. In this experiment, unsorted files were read, merged and a single sorted output file was generated. The underlying file system was NTFS. In the setup, we used 16 threads, each with 16 input files. There were 500,000 (four byte) records per input file.

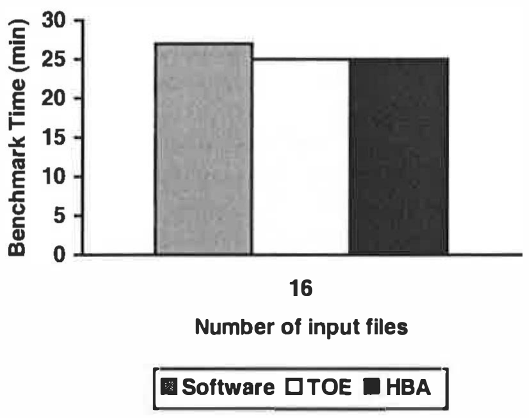


Figure 7. This figure shows the time to complete the TPIE-Merge benchmark for the three approaches: Software, TOE and HBA when the number of input merge files (each with 500,000 records) is varied from 8 to 16.

The results in Figure 7 show a small degradation (7%) in performance of the software approach with respect to the hardware approaches. The average initiator CPU utilization was close to 100% during the benchmark for all of the approaches. This particular benchmark uses the NTFS block size of 4 KB. Though the difference in the CPU utilization-to-throughput ratio for this particular block size for the hardware approaches is 33% better than that of the software approach, this difference applies to the non-application processing time. The measurements from the Perfmon tool indicate that the non-application processing time is 10-15% of the total CPU utilization and thus minimizes the impact of the reduced CPU utilization-to-throughput. The throughput requirements of this benchmark (< 10 MBps) did not approach the networking bandwidth available or the adapter processing limits. This shows that scientific applications that run on typical file systems with smaller block sizes may not get the expected performance improvements with the hardware approaches. However, a recent study of the Direct Access File System presents results from this TPIE merge benchmark that shows hardware support can improve performance by as much as a factor of two when the communication overhead is a large component of overall processing [Magoutis02].

5. Discussion

This section summarizes the performance characteristics of the software and hardware approaches based on the results obtained in Sections 3 and 4.

5.1. Software Approach

The software approach achieves the best numbers in terms of throughput and latency compared to the hardware approaches. The software approach has the disadvantage of high CPU utilization-to-throughput ratio for large block sizes as a result of high per-byte overheads. At the same time, the performance of the software approach scales with CPU speed and current CPU speeds are high enough to absorb the overheads for Gigabit Ethernet networks. In summary, the software approach is very competitive with the current generation of hardware approaches in the block sizes typically used by database and email applications and does not suffer from hardware performance bottlenecks. Consequently, the software approach shows superiority in such application benchmarks.

5.2. Hardware Approaches

The designers of the hardware approaches do achieve lower CPU utilization-to-throughput ratios for large block sizes (64 KB), but the benefit of this reduction reduces significantly when the block size is not high (0.5 KB to 4 KB). Moreover, the latency and throughput analysis points to a hardware bottleneck in protocol processing primarily because of the disparity in the processing speeds of the host system and the hardware offload. Also, the hardware approaches are not sensitive to increases in CPU speeds, because the critical processing has been offloaded to the adapter.

The above phenomenon has two implications. First, all improvements in the performance of the hardware approaches must come from increasing the processing speeds of the hardware offload through superior ASIC technology. Second, the increase in processing speed of the hardware offload must keep up with the corresponding increases in general-purpose processors so as to be competitive with the software approach. However, cost, heating, power and area constraints make this challenge harder than that in general purpose CPUs. Thus, it may be possible to decrease the parity between the processors on the host and the adapter by using superior technology, but then the resultant product may not have the right cost-performance ratio in a highly competitive storage market.

However, the hardware approaches may be well suited for high-end environments. Proponents of the hardware approaches argue that the hardware acceleration is more effective at 10 Gbps networks. The impact of communication overhead is so pronounced at such speeds that current CPUs will not be able to take advantage of the capacity of the network in the software approach. Another advantage for the hardware approaches is in high-end storage subsystems that have to support large numbers of initiators (32-1024). In the multi-initiator scenario, any reduction in the CPU utilization-to-throughput ratio can allow the storage subsystem to support a larger number of initiators. Furthermore, these high-end storage subsystems also have multiple adapters (32-64) that allow the adapters to have processing power comparable to that of the host subsystem.

The hardware approaches are also well suited for those applications where the communication overhead is a significant component of total CPU utilization. In such cases, using hardware approaches with their better CPU utilization-to-throughput ratio can significantly reduce the overhead. The benchmarks in Section 4 do not cover this category of applications.

Even so, this study is useful in designing the next generation of hardware adapters. An alternative to the current trend in the hardware approaches would be to examine the division of protocol processing between software and hardware so as to better take advantage of the superior processing power in general-purpose computing.

6. Related Work

Rod Van Meter et al [Hotz98, VanMeter98] was one of the early proponents of the concept of storage over IP. The VISA (Virtual Internet SCSI Adapter) infrastructure implemented a SCSI transport layer and aimed to demonstrate that Internet protocols could serve as a communication base for SCSI devices. The initial performance analysis identified CPU overhead as well as protocol de-multiplexing as potential bottlenecks. Around the same time, Garth Gibson et al [Gibson97] proposed two innovative architectures for exposing storage devices to the network for scalability and performance. The NetSCSI architecture envisaged exposing SCSI devices to the generic network while the NASD architecture involved exporting secure object storage services over the network.

Numerous performance studies have been conducted in the past examining TCP/IP stack overheads for gigabit networks. Keng et al [Keng96] and Chase et al [Chase01] evaluated TCP/IP at near-Gigabit speeds to provide a breakdown of the various TCP/IP costs. Their studies point out that lack of zero-copy and checksum offloading impact TCP/IP performance at high speeds because these operations increase the host CPU utilization. The implementation used in these papers uses page re-mapping techniques to implement zero-copy functionality. However, page re-mapping has alignment issues and incurs virtual memory mapping overheads, particularly in SMP environments. Also, page re-mapping does not generalize to upper-layer protocols like iSCSI.

The effect of large frame sizes on TCP/IP performance has been also well studied before by Chase et al [Chase01]. The authors show that a large frame size reduces the number of interrupts and per-packet overheads to improve TCP/IP performance. As has been pointed out in Section 2.1, large frame sizes are currently not a viable alternative in Ethernet environments due to lack of standardization. Many network adapters provide interrupt suppression features to reduce interrupt overhead even for standard Ethernet frames. In particular, the TOE approach allows for interrupt coalescing even for standard Ethernet frames by offloading the TCP/IP stack.

Magoutis et al [Magoutis02] performed a thorough analysis of the key architectural elements of DAFS [DeBergalis02]. The study reported results comparing the DAFS and NFS-nocopy protocols that utilize different zero-copy receive mechanisms. The analysis shows that while both DAFS and NFS-nocopy can achieve high throughput, the direct data placement architecture of DAFS results in lower CPU utilization. In contrast, the performance analysis in this paper complements the DAFS study by focusing on block storage protocols and incrementally examining the effect of TCP/IP offload and zero-copy support on the same protocol using the TOE and HBA approaches. The incremental analysis is useful because of the emergence of TOE adapters that do not have support for zero-copy receives for iSCSI.

Wee Teck Ng et al [Ng02] performed an exhaustive performance analysis of iSCSI over a wide-area network. This analysis proposes techniques for reducing data access times to combat the high latency in such long-haul networks. The focus of our paper is on the high-bandwidth low-latency local area network environment, and we assess the impact of the various approaches to IP storage to maximize throughput and minimize latency. Thus, the results of the wide-area network analysis are also complementary to our study.

Boon Ang et al [Ang01] did a performance investigation on the impact of TCP/IP offload. Their study indicates that if the TCP/IP offload technology uses inexpensive network processors instead of ASICs, then the resulting performance may not be able to reach wire speeds.

There is also interest in low-overhead IP storage networking for network file systems as well [Magoutis02]. This approach uses network adapters following the RDMA approach specified in Section 2.1. This study is based on a single-vendor interconnect technology whereas the conclusions in this study are applicable to network file systems using RDMA over IP networks [DeBergalis03].

7. Conclusions

This paper presents an analysis of the software, TOE and HBA approaches to build an IP storage area network. The software approach is based on the unmodified TCP/IP stacks that are part of a standard operating system distribution. For the two hardware-based approaches (TOE, HBA), we experimented with a range of adapters and chose a representative adapter for the current generation of each of the hardware approaches. The goal of the analysis is to answer the question whether current generation of hardware support necessarily helps performance.

The micro-benchmark analysis reveals that hardware support reduces the CPU utilization-to-throughput ratio for large block sizes. However, at the same time the current generation of hardware support can itself be a performance bottleneck that can hurt throughput and latency. This result is also supported by the macro-benchmark analysis that shows that the hardware approaches do not provide performance benefits in database, scientific and email benchmarks compared to the software approach, even though the hardware approaches have the potential to provide benefits in CPU-intensive applications. The analysis in this paper points out that a disparity in the processing power between the host and the adapter is the primary cause of the performance bottleneck in the current generation of the hardware approaches. This points to the need for an introspection of the current trend in hardware support so as to take advantage of the increased computing power in general-purpose processors.

Acknowledgments

We would like to thank our shepherd Jeff Chase as well as the anonymous reviewers for their valuable feedback regarding the quality of the draft submission. We would also like to thank John Hufferd, Nirmala Venkatramani and Matthew Lietzke for their invaluable assistance in setting up the experimental framework. We would finally like to thank Moidin Mohiuddin and Honesty Young for their active encouragement of our work.

References

[Ang01] Boon S. Ang, "An Evaluation of an Attempt at Offloading TCP/IP Protocol Processing on to an i960RN-based iNIC", Hewlett-Packard Technical Report HPL-2001-8, 2001.

[Ansi90] HIPPI, ANSI Standard X3T9.3/90-043, 1990.

[Bailey02] S. Bailey, "The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) On Internet Protocols", Internet Draft, IETF 2002, Work in Progress.

[Benner96] A. Benner, "Fibre Channel: Gigabit Communications and I/O For Computer Networks", McGraw-Hill, 1996.

[Chase01] J. Chase, A. Gallatin and K. Yocum, "End-System Optimizations for High-Speed TCP", IEEE Communications, 39:4, pp 68-74, 2001.

[Compaq97] Compaq, Intel, Microsoft, "Virtual Interface Architecture Specification", v1.0, December 1997.

[DeBergalis03] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, M. Wittle., "The Direct Access File System", FAST 2003.

[Gibson97] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, E. Feinberg, H. Gobyoff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, J. Zelenka., "File Server Scaling with Network-attached Secure Disks", Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems, June 1997.

[Hotz98] S. Hotz, R. Van Meter, and G. Finn, "Internet Protocols for Network Attached Peripherals", 6th IEEE/NASA Conference on Mass Storage Systems and Technologies, 1998.

[Intel02] Intel Corp., "Iometer Performance Analysis Tool", http://www.intel.com/design/servers/devtools/iometer

[Katcher97] J. Katcher, "PostMark: A New File System Benchmark", Technical Report TR3022, Network Appliance Inc, 1997.

[Keng96] Hsiao Keng, and J. Chu, "Zero-copy TCP in Solaris", USENIX 1996 Annual Technical Conference, pp 253-264, 1996.

[Kent98] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, 1998.

[Kronenberg86] N. Kronenberg, H. Levy and W. Stecker. "Vax-Clusters: A Loosely Coupled Distributed System", ACM Transactions on Computer Systems, 4:2, pp 130-146, 1986.

[Magoutis02] K. Magoutis, S. Addetia, A. Federova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremsinghe, E. Gabber, "Structure and Performance of the Direct Access File System", USENIX Technical Conference, pp 1-14, 2002.

[Magoutis03] K. Magoutis, S. Addetia, A. Federova, M. Seltzer, "Making the Most out of Direct-Access Network Attached Storage", FAST 2003.

[Ng02] Wee Teck Ng, H. Sun, B. Hillyer, E. Shriver, E. Gabber, B. Ozden, "Obtaining High Performance for Storage Outsourcing", FAST, pp 145-158, 2002.

[Rajagopal02] M. Rajagopal, E.Rodriquez, R. Weber., "Fibre Channel Over TCP/IP", Internet Draft, IETF, 2002, Work in Progress.

[Sarkar02] P. Sarkar and K. Voruganti, "IP Storage: The Challenge Ahead", 19th IEEE Symposium on Mass Storage Systems, pp 35-42, 2002.

[Satran02] J. Satran, K. Meth, C. Mallikarjun, C. Sapuntzakis, E. Zeidner, "iSCSI", Internet Draft, IETF, 2002, Work in Progress.

[Shanley02] T. Shanley and J. Winkley, "Infiniband Network Architecture", Addison-Wesley, 2002.

[TPC97] TPC Benchmark C Standard Revision 3.3.2, Transaction Processing Performance Council. 1997.

[VanMeter98] R. Van Meter, G. Finn, and S. Hotz, "VISA: Netstation's Virtual Internet SCSI Adapter", ASPLOS 8, pp 71-80, 1998.

[Vengroff96] D. E. Vengroff and J. S. Vitter, "I/O-Efficient Scientific Computation using TPIE", IEEE Conference on Mass Storage Systems and Technologies, pp 553-570, 1996.

[Voruganti01] K. Voruganti and P. Sarkar, "An Analysis of Three Gigabit Storage Networking Protocols", pp 259-265, IPCCC 2001.

More than an interface — SCSI vs. ATA

Dave Anderson, Jim Dykes, Erik Riedel

Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
{dave.b.anderson,james.e.dykes,erik.riedel}@seagate.com

Abstract

This paper sets out to clear up a misconception prominent in the storage community today, that SCSI disc drives and IDE (ATA) disc drives are the same technology internally, and differ only in their external interface and in their suggested retail price. The two classes of drives represent two different product lines aimed at two different markets. In fact, both classes contain a range of products that address a variety of features and usage patterns beyond simply the interface used to talk to the device. The target market and final product specification are taken into account from the earliest design decision through the manufacturing and testing process. This paper attempts to clarify the differences by illuminating some of these design choices and their consequences on final device characteristics. This will hopefully allow the community to build better storage systems with better knowledge of the trade-offs being made and the performance characteristics that result.

1 Introduction

Every manufacturer has different product families aimed at different customer segments. A Smart city coupe from DaimlerChrysler is much different than a Mercedes E-class sedan, although the apparent technology (gasoline engine, four round wheels) may be quite similar.

The disc drives traditionally sold with personal computer systems are quite distinct in appearance, performance and cost from those sold on larger computer systems. We will refer to the former as personal storage (PS) and the latter as enterprise storage (ES).

There are, of course, more than two classes of disc drives. Portable computers and some consumer electronics devices use disc drives that differ in important ways from either of the classes we will discuss here. We will leave as future work comparing the unique features of those drives with their larger cousins.

1.1 ATA versus SCSI

The question addressed in this paper is often phrased in terms of ATA drives versus SCSI drives. This is not accurate, as we will see: *the ATA versus SCSI debate groups the drives by interface, but the interface is perhaps the least significant difference.* Differences in mechanics, materials, electronics, and firmware make for the real distinctions among drive families and product lines. When choosing a drive for a particular application, system designers must consider these underlying factors, and not assume that the interface distinction alone is sufficient.

The interface difference may appear to categorize the drives correctly, but, in fact, does not. There have been several instances of PS drives equipped with a SCSI interface and ES drives are also used in high-end personal computers. There is no inherent reason why an ES drive could not have an ATA interface.

1.2 Personal storage

The most important quality in PS drives is that a drive have a cost commensurate with the cost of the system in which it is installed. The cost pressure of the personal computer market gave rise to the first low-cost hard discs, and has continued to put pressure on PS drive pricing. As we discuss PS drives, we will come back to this point repeatedly: low cost dominates the design of PS drives.

When the first personal computers appeared, none had a hard drive. The drives of the day were too big and far too expensive. The customer demand for a hard drive based personal computer drove the development of a small-sized, low cost drive.

1.3 Enterprise storage

Since their invention disc drives have been used on large computer systems. At the time, these systems tended to be very big, expensive and were employed to access large quantities of data. Because of the cost, they were used to support many users simultaneously.

This environment gave rise to the essential properties of ES drives. First, they tend to be configured in groups (aggregation), as opposed to PS drives, which are most often the only drive in a system. Second, they are used to randomly access small portions of large data spaces. Third, reliability and performance are critical characteristics. A failure could idle a considerable number of employees and directly impact business operations. In normal operation, the faster the drives can service requests, the more employees can be supported and the more productive those workers can be.

1.4 Key requirements

We will now look at these key requirements and see how they have manifested themselves in drives for each market.

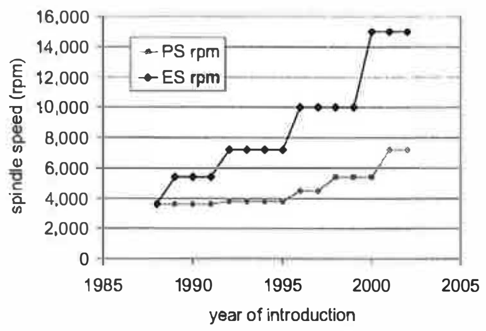


Figure 1: The adoption of higher rotational speeds. Data from Control Data product guides, 1988-1989 and Seagate product guides, 1990-2002.

1.4.1 Cost

There is constant pressure to reduce drive costs, even as drives become more complex to build. Due to the resulting demands on encoding schemes, error correction, and servo processing, it takes considerably more logic to control basic reading and writing with every areal density improvement. It also requires greater precision and lower tolerances for noise and interference of any kind. Each component of a drive must become more complex in order to deliver state-of-the-art capacity, while at the same time being pushed to become less costly to build.

1.4.2 Seek performance

Improving seek performance is the continuous struggle to get the head to move from one location to another faster than in the previous generation product. This involves using more expensive components such as higher performance magnetic circuits, faster microprocessors and lower-mass actuator assemblies. The process of designing an ES drive involves more sophisticated modeling and analysis to optimize the structures for seek movements. The various vibrational modes of the structure can negatively affect seek performance. Fast seeks depend on the ability to rapidly follow the servo patterns on the media in a predictable way. The design must preclude drive seeking being throttled by an obscure resonance of the head/disc assembly [IBM99c].

1.4.3 Rotational latency

Latency is improved by spinning the media faster. PS drives are much slower to adopt the performance improvements first introduced in ES drives. PS performance enhancements are made only when they do not incur any marginal cost. After a given capability has been in ES drives for some years it is practical to move it to PS models; the cost penalty and development cost having been eliminated by the volume of ES market. Figure 1 shows the history of rpm adoption in mainstream products over the last 15 years.

In fact, this history illustrates a general characteristic of the relationship between ES and PS drives. ES drives tend to drive costly innovation - achieving new levels of performance, reliability or function - and PS drives adopt that technology when it becomes cheap enough. This is a model that puts ES drives in a difficult pricing position compared to PS drives, but growth in the ES market depends on these added capabilities.

There is innovation in PS drives as well, but it tends to be in terms of cost savings, such as making a 7,200 rpm motor cheaper, rather than building a 15,000 rpm motor for the first time. ES drive cost comes in the form of higher cost of materials, but also in larger research and development investment.

1.4.4 Aggregation

A notable difference in operating environment between PS and ES drives is the use of ES drives in groups. This is more than simply an interface issue - just being able to electrically interconnect multiple drives. A property of Fibre Channel (FC), SCSI and Serial Attached SCSI (SAS) is that they efficiently attach more drives to a host than the two drive limit of a traditional IDE controller.

That is not, however, all there is to aggregation. If drives are housed together and used at the same time, interactions occur that can dramatically decrease performance if no compensation is included. When one drive is trying to seek or simply stay on track while nearby drives are spinning, there is an energy transfer, known as *rotational vibration*, from one seeking drive to the other drives in the cabinet.

1.4.5 Reliability

Reliability varies significantly with usage patterns and operating environment. Personal computers are designed for active use only several hours per day, while most enterprise systems are active 24 hours a day, every day. This means that design choices made in PS drives for cost reasons will make them less likely to perform well under operational stresses for which they were not designed.

1.5 History of the interfaces

Traditionally, the difference in the two interface was based on how much work was done by the host and by the drive. Until a few years ago, IDE controllers used programmed I/O, where the main system processor was responsible for all interactions with the disc drive, without interrupts or direct memory access (DMA) to offload data transfer. In SCSI, there was always an external control chip on the drive that handled independent operation of the drive.

While a standards group is currently adding a command queuing function similar to that in SCSI to the Serial ATA (SATA) protocol, ATA historically has not added any of the major features of SCSI: multiple CPU support (both failover and simultaneous operation), variable block size support (that is, the ability to specify and format the drive to a non-512 byte block sector) and dual porting. Note that as this type of ES drive functionality accretes to PS drives, the complexity of implementation also increases.

2 Technology Differences

The differences between PS and ES drives are far-reaching and start from the earliest design choices. The diagram in Figure 2 illustrates the basic components of a modern disc drive. This section will consider each of these items in turn. Note that since the market for disc drives is a very cost-sensitive one, drive designers will not spend an extra penny in material or assembly cost to go beyond the target device specifications.

2.1 Mechanics

The basic component choices in the mechanical portion of the drive affect the overall reliability, seek time, acoustics, and resistance to temperature, shock, vibration, and other environmental variations.

2.1.1 Head/Disc Assembly

The head/disc assembly (HDA) consists of the base casting, heads, actuator, spindle, discs, air handling system, and top cover. The ES drive operates at higher rpm, while also maintaining a higher tolerance for external disturbance. These external disturbances could be the influence of neighboring drives — rotational vibration — or other environmental factors such as temperature. This is complicated by the fact that higher rpm and faster seeking ES drives put more energy into a drive cabinet, creating more disturbance. At the same time the drives are required to be less affected by it. This requires more rigidity in the mechanical structure of the drive, more mass, higher bandwidth servos, and in some cases special support circuitry to offset effects that could otherwise decimate a drive's performance.

Higher rpm drives also require more power to operate, creating more heat that can affect the drive or its neighbors in a cabinet.

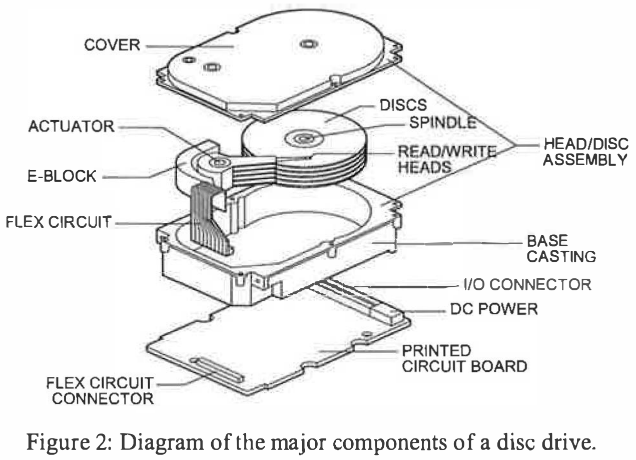


Figure 2: Diagram of the major components of a disc drive.

Achieving a million hour MTBF drive is not an easy thing. Every failure mode must be addressed. ES drives will have tighter tolerances and design rules to control externally and internally generated particles and outgassing. These rules include such things as avoiding through holes, greater environmental control and higher quality sealing. The ES drive typically has more environmental protection. An ES drive will have a filter for particles, a desiccant to control humidity, and an active carbon absorbent for eliminating organic substances inside the HDA. The spindle motors have O-ring seals, and the drive cover better gasketing. Each of these little things adds cost but improves reliability. Individually, each one addresses a relatively minor failure mode, but together they help achieve 1,000,000+ hour MTBF.

PS drives are designed for reliability, but they tend to compromise where components can be eliminated to save cost. The O-rings and desiccant, for example, are usually eliminated in PS drives.

An ES drive has more shrouding and air control devices to better manage air flow inside the HDA. This eliminates air turbulence, which would otherwise make it harder to keep the head on track and optimize seek performance. It also directs air to the actuator to help cool it. It also adds cost.

The size and stiffness of the base casting and top cover impact both the acoustic characteristics of the drives, as well as the susceptibility to rotational vibration. Both of these problems become more acute at higher spindle speeds.

2.1.2 Actuator

Larger magnets are key to achieving faster seek times, but they bring additional requirements, along with a higher cost. In order to get the most seek performance and still stay within a tight power budget, the ES actuator coils must have less resistance. This requires thicker coil material with fewer windings. As already mentioned, special HDA design features promote cooling the actuator to prevent overheating.

An interesting example of complex interactions arises with the latch. Inside every drive is a latch to hold the actuator when power is off. The most common method of latching involves a magnetic circuit. However, the latch has a magnetic field associated with it, which can affect seek performance when the actuator is operating near the latch. In a PS drive, there is no compensation for this, as seek performance is not critical. To achieve the optimum seek performance, ES drives will have a bi-stable latch that does not affect performance. This is a more expensive solution, but gives better overall performance.

Both the coil and the bearing cartridge are independently bonded to the arm using a special epoxy in an ES drive. In a PS drive the coil is likely to be attached to the arm with a single molded connector, a less expensive technique. The former makes for a more rigid structure and is necessary to achieving maximum seek performance.

In a PS drive, seek performance is not the high priority it is in an ES drive. Typically a PS drive design must first achieve its cost targets, and then do the best it can with seek performance. The opposite priority holds with ES drives, e.g., the actuator design must prevent its various bending modes and resonances from impacting seek, settle time, and performance in the presence of rotational vibration.

2.1.3 Spindle

For over 15 years drives spun no faster than 3,600 rpm. Since then drives have been sped up first to 5,400 rpm, then 7,200 rpm, 10,000 rpm and most recently 15,000 rpm. Spinning faster is a tremendous engineering challenge. The read/write head must be kept on track, and this is increasingly difficult as rpm goes up. An off-track head during reading can cause a mis-read and a *rotational miss* (requiring a full rotation before the read can be re-tried). An off-track head during writing can cause a mis-write that introduces noise or even overwrites adjacent tracks.

Higher rpm requires more expensive motors. As tracks per inch (TPI) increases the motor becomes a bigger challenge. Disturbances such as *windage* (air movement between the disk and arm) and vibration increase with rpm. At the same time ES drives must be less affected in order to get the best possible random performance. For cost reasons PS drives use a cantilever motor design, where the motor shaft is captured only at the base deck end. An ES motor shaft is captured at both ends, with an attachment to the top cover. With today's TPI, fluid bearing motors are preferred since they minimize runout and acoustical noise (see below for a discussion of runout). For years it was thought impossible to have a fluid dynamic bearing motor captured at both ends. Seagate solved this with a unique conical design that gives ES drives the benefits of both fluid bearings and a motor supported at both ends. This is a more expensive design, but gives better overall performance.

2.2 Electronics

The on-drive electronics are becoming more integrated as improvements in processor technology allow [Matsumoto99]. This means that fewer components are required to provide the same basic functionality.

2.2.1 Control processor

The drive servo system keeps the read/write head on track or moves it from one track to another. The drive determines its position by reading very small fields of information interspersed among the data blocks on every track (servo bursts). Every time the head crosses over a servo burst, the microprocessor suspends what it is doing and takes up the task of identifying where the head is. If it is wandering off track slightly, it must move the head in the appropriate direction and distance to get back in the middle of the track. During seeks, the actuator constantly reads servo bursts as it crosses tracks. This information is used to determine how close the actuator is getting to the target location and, when it is close, to decelerate the actuator.

2.2.2 Servo processor

As TPI gets higher, more servo processing is needed to keep the head off neighboring tracks. This would not be so hard if the tracks were perfect, repeatable circles. They are not: motor variation, platter waviness (both circumferentially and radially), stacking tolerances and other factors give rise to both repeatable and non-repeatable runout. *Runout* - variation in the radius or circumference of the track - occurs when the head is unable to follow the current track and stay in position above it. Repeatable runout is inherent in the track, and is the same on each rotation, making it easier to compensate for. Non-repeatable runout is due to external influences such as vibration, and varies over time. The servo processor must adjust the head to follow the track wandering underneath it. To get more servo capability, higher capacities require more servo bursts. This requires more processing works against minimizing cost and increasing capacity. A PS drive is a constant balancing act between minimizing cost - including processing power - and tracking the higher TPI's to achieve maximum capacity.

2.2.3 Interface

There is significantly more silicon on ES products. The following comparison comes from a study done in 2000:

· the ES ASIC gate count is more than 2x a PS drive,

· the embedded SRAM space for program code is 2x,

· the permanent flash memory for program code is 2x,

· data SRAM and cache SRAM space is more than 10x.

The complexity of the SCSI/FC interface compared to the IDE/ATA interface shows up here due in part to the more complex system architectures in which ES drives find themselves. ES interfaces support multiple initiators or hosts. The drive must keep track of separate sets of information for each host to which it is attached, e.g., maintaining the processor pointer sets for multiple initiators and tagged commands.

The capability of SCSI/FC to efficiently process commands and tasks in parallel has also resulted in a higher overhead "kernel" structure for the firmware. All of these complexities and an overall richer command set result in the need for a more expensive PCB to carry the electronics.

When the drive processor is busy doing servo work and read/write tasks, it cannot be doing interface work. In order for an ES drive to offer the maximum performance, it is equipped with two processors - one dedicated to servo and the other for interface and read/write handling. Maximizing random access and performance under rotational vibration both depend on that dedicated servo processor.

A PS drive has a single processor, which must handle all three basic processor tasks in a drive. It must run the interface, support the reading and writing of data and do all the servo processing.

2.2.4 Memory

The firmware for the SCSI command set is more than twice as large as that for ATA, requiring more permanent flash for code and increased SRAM at runtime. The more complex command set and larger command queues also require additional memory space. The SCSI command set allows for vendor-specific extensions which require additional code space, allowing greater flexibility in configuration.

2.3 Magnetics

In magnetic componentry there is much similarity between the ES and PS drives since both strive to stretch the same areal density boundary. Differences stem from the performance goals of the ES drives. The higher rpm of ES drives delivers higher data rates.

2.3.1 Heads

Though magneto-resistive head technology has made a profound change in how data is read in a drive, writing is still an inductive process. As such it is sensitive to linear velocity and higher rpm improves not only latency, but data rate, as well. For this reason, ES drives tend to stretch writing capability, and demand constant innovation to keep up with the high rpm and higher areal density. PS drives usually adopt the writer technology proven in previous generations of ES drives.

Reading is just the opposite. Read data rate is generally insensitive to linear velocity, but in some cases it may be adversely affected by higher rotational speed. Signal amplitude does not increase as it does with inductive heads, but noise does. This means that ES drives, with their higher rpm and data rate targets, have a more difficult magnetic environment in which to read data.

The key property to having a system that will read and write reliably is the signal to noise ratio (SNR). It is much harder to reach a given SNR in a high rpm drive. This makes the task of extracting the data from the read signal significantly more difficult. This is sometimes referred to as recording stress, and is usually more pronounced in an ES drive. ES drives must have more expensive read/write electronics to cope with this more difficult magnetic environment and higher data rate.

2.3.2 Materials

The traditional substrate material for media is aluminum, onto which a layer of magnetic material is deposited. The recent use of glass substrates provides a greater uniformity of the magnetic surface and greater stiffness [IBM99], but the magnetic layer is harder to deposit on glass, making it more difficult and expensive to achieve the same read densities [Walker01]. The better shock tolerance of glass must be traded against lower density or data rate. In addition, since glass cannot be textured, the actuator must be removed from the media to land (load/unload ramp) rather than landing on the media (contact start/stop) [IBM99a]. This requires a landing zone at the outer edge of the disc in case there is contact as the heads leave the platters - precisely the area of highest density and data rate.

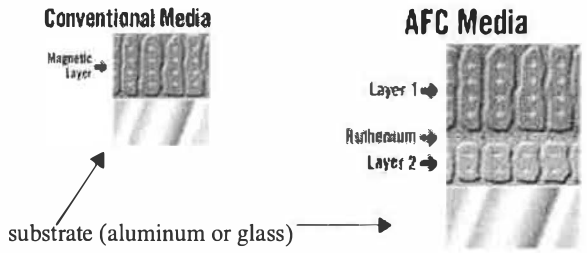


Figure 3: Diagram of media layers. The base substrate consists of either aluminum or glass, topped with a layer of magnetic material. In anti-ferromagnetically coupled (AFC) media, an additional layer of magnetic material and a layer of ruthenium are added, with the two layers reinforcing each other for better magnetic stability at higher density.

A recent change in media structure is the use of anti-ferromagnetically coupled media, which contains a second magnetic layer oriented opposite the primary layer to reinforce the magnetic orientation [IBM01]. This is necessary to achieve higher densities, at the cost of increased complexity in both materials and in the manufacturing process [Walker01]. The diagram in Figure 3 illustrates this layering.

2.4 Manufacturing

The build and test times for ES drives are considerably longer than PS drives. Increased test time can make a drive more reliable. During this time, drives also undergo detailed characterization, such as learning precisely how irregular individual tracks are, which allows them to better keep the heads on track during normal operation. More time spent analyzing the media for flaws results in lower probabilities these flaws causing unrecoverable read errors in the field.

3 Performance Differences

We have outlined the design choices possible when designing a disc drive for a particular target market. Most of these choices affect performance in some way, and we will now attempt to quantify the impact of specific choices.

3.1 Capacity

The basic media structures used are the same in both drive types, with the highest areal density used at any given time. The choice of the number of disc platters and the size of the platters changes the overall capacity - for example, 15,000

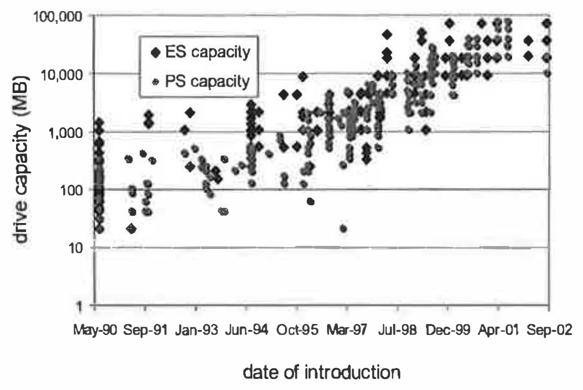


Figure 4: Comparison of capacities. Capacity and introduction date of 10 years of Seagate drives [Seagate02].

rpm drives use 2.5" platters to support the faster spindle speeds, while 7,200 rpm drives use 3.7" platters.

3.1.1 Size of Platters

ES drives spin faster to get better performance. However, power increases almost to the cube of rpm. Smaller diameter platters keep drive power at an acceptable level. This has a cost: an ES drive uses more platters to achieve the same capacity as a PS drive at a given areal density. The smaller platters actually brings two performance advantages - the ability to spin faster and faster seeking. Average seek times are better because the head must traverse a smaller recording band. This, together with the greater investment in actuator capability as discussed earlier, makes for drives that perform random access much faster than their PS counterparts at equivalent areal densities.

The larger diameter platters and the lower rpm give the PS drive a clear advantage in delivering capacity. This is consistent with the primary market requirement of lowest cost. A combination of minimizing the parts cost and delivering the highest capacity yields the lowest dollar per gigabyte. The data in Figure 4 compares drive capacity against date of introduction over the last 10 years.

3.1.2 Number of Platters

Many drives are manufactured with fewer platters than possible, as performance matters more than capacity. The chart in Figure 5 illustrates this trend toward depopulated drives, as well as the more recent use of depopulated heads (only using one platter surface to save the cost of the additional read/write head).

Fewer platters translates into faster seeks because there are less heads, so the actuator has a lower total mass and can move a fraction of a millisecond faster, which can be significant at sub-4 ms average seek times. This also matches the marketplace as users with a requirement for performance will often buy more drives, each of a lower capacity, to spread data across as many actuators as possible.

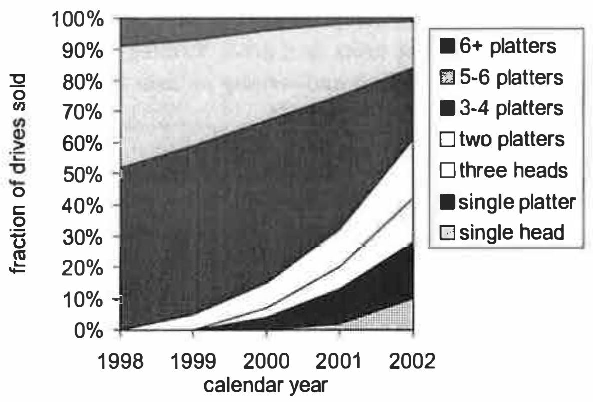


Figure 5: Trend toward depopulated drives. More users are choosing drives with less platters, trading capacity for performance.

3.2 Data rate

The fastest ES drives will always have higher data rate than a contemporary PS drive, due in large part to the higher rpm, as explained earlier. However, the PS drive has an advantage in media size. Typically PS drives use 95 mm (3.7") platters, compared to 84 mm (3.3") in 10,000 rpm and 65 mm (2.5") in 15,000 rpm ES drives respectively. The larger media size helps the PS drives follow closely in data rate.

Another factor favoring the PS drive is that new models tend to come out more frequently than ES drives. Introduction of a new ES drive comes when the new generation is able to double the capacity of the previous generation. Hence successive models over the last several years have been 9, 18, 36, 73 and 146 GB. PS drives, on the other hand, come out as soon as it is possible to deliver an appreciable increase in capacity. Instead of doubling, they have been introduced at 10, 20, 30, 40, 60, and 80 GB per platter. This higher frequency enables PS drives to stay much closer to ES drives in data rate than if they were following the same "jumps" that happen for ES drives. The data in Table 1 compares the data rates of several drives and shows the underlying spindle speed, areal density, and platter size.

		cap	**speed**	**density**	**dia**	**int bw (Mb/s)**		**ext bw**
		GB	rpm	Gb/in^2		calc	spec	MB/s
Atlas 10k 18WLS	ES	18	10000	3.4	3.3"	-	314	24.6
DeskStar 75	PS	30	7200	11.0	3.7"	551	444	35.6
Cheetah 36LP	ES	18	10000	7.3	3.3"	579	427	-
Cheetah X15	ES	18	15000	7.3	2.5"	690	508	39.5
Cheetah X15-36LP	ES	36	15000	17.5	2.5"	969	709	57.7

Table 1: Comparison of drives with increasing data rates. Capacities, speeds, and densities are from published spec sheets. Diameters are typical for those spindle speeds. Internal bandwidths are calculated from the speed, diameter, and TPI as shown in the spec sheets. External bandwidths are as measured by LinuxHardware.org [Augustus01].

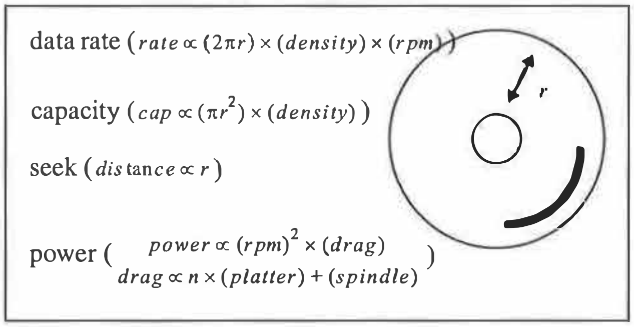

data rate ($rate \propto (2\pi r) \times (density) \times (rpm)$)
capacity ($cap \propto (\pi r^2) \times (density)$)
seek ($distance \propto r$)
power ($power \propto (rpm)^2 \times (drag)$; $drag \propto n \times (platter) + (spindle)$)
r

Figure 6: Diagram of basic drive parameters. A smaller media (lower r) sacrifices bandwidth and capacity for shorter seeks and lower power. Each additional platter adds to capacity and power consumption.

This table shows the three components to sequential data rate: the rotational speed, the areal density, and the diameter of the platters. Higher speed drives will use smaller platters for lower energy consumption and faster seeks, resulting in lower data rates. The five drives in the table are arranged in order of externally-measured sequential throughput. We see that the 7,200 rpm DeskStar is faster than the 10,000 rpm Atlas due to a much higher areal density, and a larger platter diameter. The DeskStar and the Cheetah 36LP are quite close in data rate because the increased rpm of the Cheetah is only enough to overcome the density disadvantage and the smaller platter diameter. The Cheetah X15 at 15,000 rpm gains data rate, but loses some due to the further reduced platter diameter. Finally, the second generation Cheetah X15 increases the areal density and far outperforms the others even with the smallest diameter platter.

The diagram in Figure 6 illustrates the trade-off among data rate, capacity, seek time and power consumption when choosing a platter size.

3.3 Random performance

Random performance describes the ability of a drive to get from one location to some other unpredicted address to service the next request. There are three components to the performance of this movement - seek performance, controller overhead, and rotational latency.

3.3.1 Seek times

Several of the mechanical items mentioned in the last section directly affect the ability of the drive to seek quickly and to stay on a servo track in response to environmental factors. The data in Figure 7 compares the seek time of drives against their date of introduction.

Seek performance of PS drives always lags that of ES drives, and improves at a slower rate, while ES drives are expected to squeeze out a gain with each new generation of drives. The entire mechanical design of an ES drive is focused on achieving the highest random access performance, as this is critical in the target market.

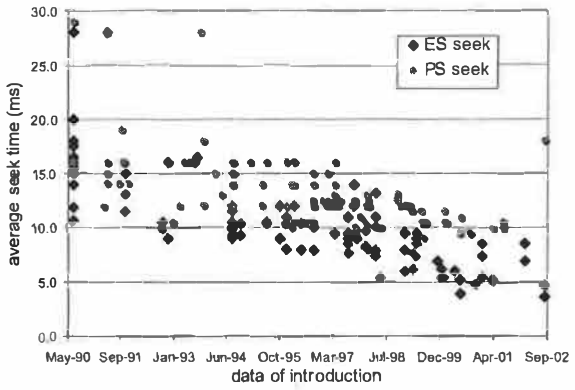


Figure 7: Comparison of seek times [Seagate02].

server	Dell PowerEdge 2550
operating system	Windows 2000 Pro SP3
scsi controller	Adaptec 39160
ata controller	Promise Ultra 100 TX2
ES drive	Seagate Cheetah 73LP - 73 GB
specs	10,000 rpm; 18 Gb/in^2; 5.1 ms seek
PS drive	Seagate Barracuda IV - 80 GB
specs	7,200 rpm; 31 Gb/in^2; 9.5 ms seek
benchmark	iometer (jan 2001 code release)

Table 3: Experimental ES and PS drive testbed.

Table 2 shows the seek performance of a PS against an ES drive under the same workload. This comparison is between a Barracuda IV and a Cheetah 73LP drive in the same system. Table 3 details our experimental setup. The mechanical details of these two drives are different, but quite close - higher density in the Barracuda compensates for the higher spindle speed in the Cheetah. The higher spindle speed will also account for some of the improvement in random performance on the ES drive.

queue depth	**read (8 KB)**		**write (8 KB)**	
	PS	ES	PS	ES
1 requests	65 req/s	115 req/s	105 req/s	184 req/s
2 requests	66 req/s	116 req/s	105 req/s	184 req/s
4 requests	71 req/s	146 req/s	105 req/s	187 req/s
8 requests	79 req/s	174 req/s	105 req/s	190 req/s
16 requests	89 req/s	202 req/s	108 req/s	200 req/s
32 requests	101 req/s	235 req/s	108 req/s	213 req/s

Table 2: Comparison of random request rates at increasing queue depth on the same request stream in PS and ES drives. Both drives are run with write caches enabled. If the write cache on the ES drive were disabled, the improvement with larger queue depth would be even larger, as observed in a previous study [White01].

3.3.2 Seek scheduling - queue depths

Seek sorting impacts performance and PS drives generally have shorter queues. In fact, the shorter queue lengths when using the ATA interface also have a direct impact on drive mechanics. The fact that seeks are not aggressively scheduled in PS drives keeps the average seek distance closer to the theoretical average of 1/3 the disc radius. ES drives with more aggressive scheduling can bring this as low as 1/10 of the radius on average. This means that the mechanical *duty cycle* - the total amount of time spent seeking and stressing the mechanical components - of the PS drives could be more than 3x higher for a similar request stream.

The data in Table 2 compares the random performance of a PS drive against an ES drive as the queue depth seen at the drive increases. With a queue of 32 pending requests, the ES drive is able to achieve more than twice the random performance possible with only a single queued request, while the PS drive only improves throughput by 55% for reads and barely at all for writes. This is similar to results in an earlier study [White01] where ES performance increase by 100% and PS performance by only 45%. The improvement comes because seeks are smaller, which leads to both better performance and better reliability. Additional scheduling sophistication could be included in the PS drive as well, but would require some of the additional electronics discussed earlier.

3.3.3 Controller overhead

Controller overhead is optimized by having as much processor performance available as possible to interpret and schedule commands as they arrive. More recently, this has been augmented with custom hardware assist to provide more performance than could be economically realized simply by greater investments in software. Such hardware ensures that data can be moved to and from the interface at rates as close to the internal drive data rate as possible.

3.4 Rotational vibration

When one drive is trying to seek or simply stay on track while nearby drives are spinning, there is an energy transfer from one seeking drive to the other drives in the cabinet. This tends to excite the drives to rotate around their center of mass, throwing the actuator off track. Unless a drive is designed to mitigate this effect, writes will abort or seeks will fail to find the desired track. In most cases this will manifest as a decrease in performance as aborted writes and rotational misses accumulate. At its extreme, this effect can get so bad that any drive, ES or PS, will cease to function. It simply could not stay on track long enough to complete any operation. The key is to understand how much rotational vibration is likely to be present in a server environment and design the drive to withstand it.

Since PS drives are built to be in single drive systems, rotational vibration is not an important factor. Though a CD-ROM drive can create a certain amount of vibration, the slight and infrequent effect is not sufficient to produce a noticeable performance problem. PC responsiveness is measured only by what a single user can see, and even a few retries would not create a serious problem in most cases.

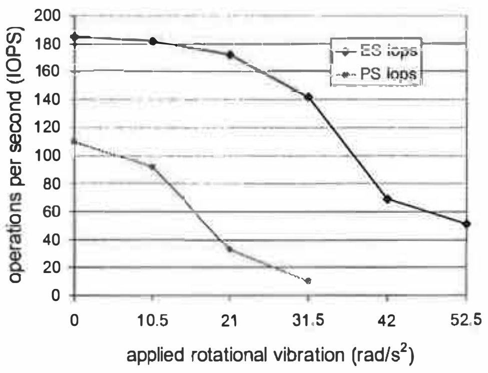


Figure 8: Externally applied rotational vibration can have a major, negative impact on performance. Individual drive cabinets vary widely in the amount of rotational vibration they transfer, and have been measured up to 45 rad/s^2 [Hall00]. Data for Seagate Cheetah 18LP and Barracuda III.

ES drives, on the other hand, are explicitly designed to operate in cabinets full of spinning drives. This requires designing a drive to maintain its operation in the presence of considerable rotational vibration. As tracks per inch (TPI) increases, the rotational vibration problem gets worse. It is more difficult to stay on track even in ideal conditions, much less with external vibrations that are difficult to compensate for [Abramovitch96]. Some recent drives have added a rotational vibration sensor that can detect external rotation and compensate in the servo processing.

Earlier we mentioned the performance degradation possible due to rotational vibration, which we will attempt to quantify here. The chart in Figure 8 shows the performance of a single drive on a test stand under varying rotational vibration. Performance of the PS drive is much more affected than the ES drive. The PS drive essentially stops at 30 radians/s^2 of external vibration, while the ES drive degrades much more smoothly and is able to operate beyond 60 radians/s^2.

When multiple drives are placed together in the same cabinet, the rotation induced by adjacent discs or other system components affects performance. The design of a cabinet and mountings determines how bad this effect will be in a particular system. Studies of more than 20 drive enclosures and machine designs from a variety of manufacturers show a wide range of vibration characteristics - from the best designs that subject the drives to 5 radians/s^2 with only minor performance consequences, through cabinets inducing up to 45 radians/s^2 [Hall00].

3.5 Reliability

One of the trickiest drive characteristics to measure is reliability, which arises from a wide range of factors and consid-

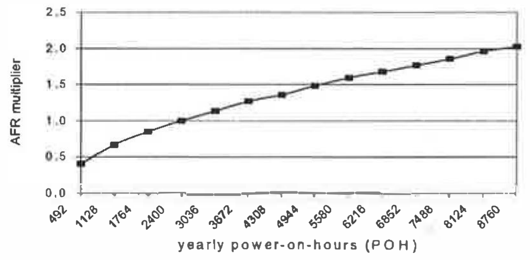


Figure 9: Reliability reduction with increased power on hours, ranging from a few hours per day to 24 x 7 operation [Cole00].

erations in design, manufacturing, and in the operational environment [Kaczeus90, Yang99, Elerath00].

The most significant difference in the reliability specification of PS and ES drives is the expected power-on hours (POH) for each drive type. The MTBF calculation for PS assumes a POH of 8 hours/day for 300 days/year[1] while the ES specification assumes 24 hours per day, 365 days per year. The longer a drive is expected to be running, the lower the MTBF, and the higher the annual failure rate (AFR).

The chart in Figure 9 shows the expected increase in AFS due to higher power-on-hours. Moving a drive from an expected 2,400 POH per year to 8,760 POH per year would increase the failure rate almost two-fold, if there were no compensation elsewhere in the design.

3.5.1 Duty cycle

In addition to the obvious increase with increased power-on hours, the amount of mechanical work the drive has to do is affected by its basic structure and by the workload it is asked to do. A larger number of platters in the drive increases capacity, but also increases the mechanical stresses.

The chart in Figure 10 shows the increase in expected AFR with higher duty cycle. The increase is higher for the drive with the larger number of platters. For a four platter disk, a duty cycle of 40% instead of 100% would reduce the failure rate by almost 50%.

Better seek scheduling leads to shorter seeks on average and therefore a lower effective duty cycle for the same set of user requests. In preliminary measurements on our testbed, we see a mechanical duty cycle of approximately 40% for the ES drive against 75% for the PS drive on the same set of requests.

Adding platters and heads increases the AFR not just due to the additional mechanical stresses, but also due to increased internal heat generation, and the additional head/disc interfaces which might release particles or lead to other negative interactions, such as head crashes.

[1]This is the specification used by Seagate, other manufacturers use varying, but similar, assumptions about power-on hours [Vilsbeck02].

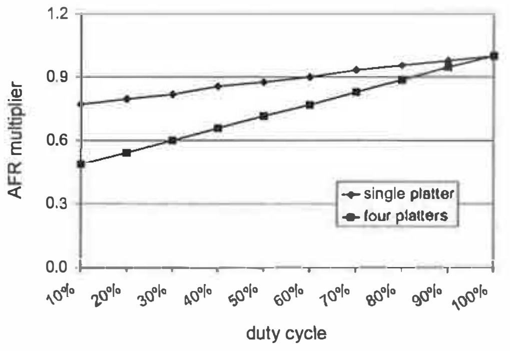


Figure 10: Reliability is decreased with higher duty cycle, and the effect is greater for drives with larger numbers of platters [Cole00].

3.5.2 Temperature

Reliability decreases with increases in ambient temperature. The drive temperature is affected not only by the outside temperature, but also by other components in the system. A high-density server rack with many disc drives grouped close together may experience much higher temperatures than a single drive mounted in a desktop computer.

The chart in Figure 11 shows increased AFR with increased temperature. A fifteen degree temperature rise is expected to increase the failure rate by a factor of two, and an increase of that size is a common assumption in high-density server racks [Patel01].

In order to prevent data corruption and failure at very elevated temperature, some drives contain temperature sensors that provide warnings of temperature outside the specification range [Herbst97].

3.5.3 Overall reliability

Each of these factors makes an individual contribution to drive failure rate, and they may also magnify each other. A capacity-focussed drive with more platters and less sophisticated seek scheduling may have a higher base duty cycle under certain workloads, and may also be more subject to temperature variation.

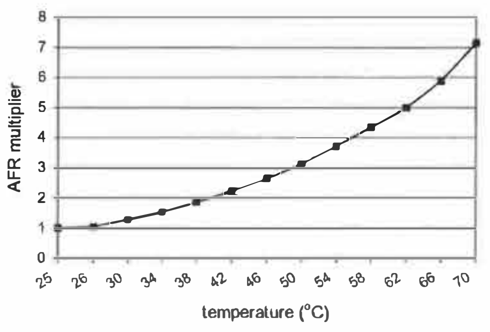


Figure 11: Reliability decrease due to ambient temperature variation [Cole00].

	iface	cap	price	speed	seek	density	kbpi	ktpi	internal bw				dia^	ext bw	dsks	cap
									3.7"	3.3"	3.0"	spec				raw
UltraStar 36LZX	SCSI	36 GB	$550	10000 rpm	4.9 ms	7.0 Gb/in^2	352	20.0	*645*	*610*	552	452 Mb/s	3.0"	36 MB/s	6	594 Gb
DeskStar 75	ATA	30 GB	$159	7200 rpm	8.5 ms	11.0 Gb/in^2	391	28.4	551	*487*	*442*	444 Mb/s	3.7"	37 MB/s	2	483 Gb

Table 4: Comparison of PS and ES drives from IBM [White01]. The Deskstar drive has a slight advantage in sequential bandwidth, even though the UltraStar has a higher rpm. The authors of the previous study attribute this to overhead in the SCSI interface. In fact, a closer look at the physical discs shows the most likely explanation a smaller platter size in the UltraStar (3.0" instead of the normal 3.7"). This reduces seek time at the expense of lower sequential bandwidth on the outer tracks. Since the UltraStar has a much lower areal density, it must also make up the capacity difference by using additional platters (6 vs. 2). ^estimated based on the internal transfer rate and raw capacity differences

	iface	cap	price	speed	seek	density	kbpi	ktpi	int bw		dia	ext bw	disks	cap
									calc	spec				raw
UltraStar 36Z15	SCSI	36 GB	$381*	15000 rpm	4.1 ms	10.7 Gb/in^2	397	27.0	798	647 Mb/s	2.6"	53 MB/s$^{\%}$	6	661 Gb
DeskStar 120	ATA	60 GB	$99	7200 rpm	8.5 ms	29.7 Gb/in^2	547	54.0	771	592 Mb/s	3.7"	48 MB/s$^{\%}$	2$^{\#}$	979 Gb

Table 5: Comparison of a newer generation of drives from IBM. In this case, the new UltraStar increases sequential performance over the new DeskStar due to the higher spindle speed, even though the areal density is lower. *from Harddrive.com in August 2002 $^{\%}$according to the published specification, not a measured number $^{\#}$the 60 GB version of the DeskStar 120 has 2 disks, but only 3 heads, one surface remains unused

Past work comparing the reliability of PS against ES drives reported a failure rate of 25% for 24 IDE drives against 2% for 368 SCSI drives over an 18 month period [Talagala99]. However, these numbers cannot be treated as a controlled study due to the very small sample size for the PS drives.

Another study using data collected during the design phase of two different drives - it is not reported whether they were SCSI or ATA drives - shows a less than 1% annual failure rate for one, and a larger than 4% for the other [Hughes02]. This clearly shows that different design choices can have a significant impact on the final drive failure rates.

4 Related Work

A previous comparison of SCSI vs. IDE [White01] concluded that IDE had slightly better sequential performance, but lagged significantly in random performance. However, the authors of that study did not compare all the mechanical details of the two drives, leading to a conclusion that cannot be generalized to all SCSI and all IDE drives.

The data in Table 4 compares a set of basic characteristics for the two drives considered in their study. The slight advantage of the ATA drive in sequential performance is due to the density advantage of the ATA drive (almost 60% higher) and the larger platter diameter (between 15% and 25% larger), which is not overcome by the rotational speed advantage of the SCSI drive (40% higher). A SCSI drive with a comparable density would perform significantly better, as discussed in Section 3.2.

The data in Table 5 shows the improvement to the next generation of both drives from the same manufacturer. In the newer SCSI drive - comparing the UltraStar 35Z15 to the DeskStar 75 at the same areal density - the rotational speed advantage, even with smaller diameter platters, still push the SCSI drive to much higher data rates (40% higher).

The advantage of the SCSI drive over the ATA drive in random performance is partly due to the smaller platters, as well as additional differences in the mechanics as explained in earlier sections. Also note that between the two generations of SCSI drives, the seek performance has improved by almost 20%, while the seek performance of the ATA drives has remained constant.

A performance comparison under Windows 2000 [Chung00] shows an IDE drive only 20% slower than a SCSI drive on sequential throughput and 44% slower on random performance. As shown in Table 6, most of this performance difference is again due to the mechanical differences. The higher density and larger platters of the IDE drive almost compensate for the faster spindle speed of the SCSI drive, although both seeks and latencies are significantly lower in the higher rpm drive. If these drives had been introduced with the same density, the higher rpm drive would also have a larger sequential throughput advantage.

	cap	seek	speed	density	dia	int bw (Mb/s)		ext bw
	GB	ms	rpm	Gb/in^2		calc	spec	MB/s
Fireball lct 08	26	9.5	5400	6.1	3.7"	343	257	19
Atlas 10K (SCSI)	18	4.5	10000	3.4	3.3"	444	314	24

Table 6: Comparison ATA vs. SCSI under Windows 2000 [Chung00].

A comparison of SCSI and ATA for end users [Dominguez99] makes many of the high-level points discussed here. ATA drives are optimized for simplicity and low cost, while SCSI drives must be optimized for performance, reliability and the ability to connect to multiple hosts. Trends in the speed and sophistication of the interfaces have been bringing ATA and SCSI closer together, with ATA gaining complexity as it moves closer to SCSI.

The comparison of ATA and SCSI reliability from the end user perspective, covering many of the factors mentioned above, was discussed extensively in a recent online article [Vilsbeck02].

Trends and recent innovation in disc drive technology, as well as the details of a specific SCSI drive design have recently been published by another disc drive maker [Miura01, Aruga01].

The design of disc drives is a very complex and multi-faceted process, and has been used as an example for students to understand engineering and cost trade-offs [Richkus99].

5 Summary and Discussion

To compare any two individual drive models at a given capacity point, one has to look at the detailed device specifications as these impact every aspect of drive design and determine drive performance. Looking at those factors in turn and comparing the impacts:

Capacity is about the same for both markets, everyone wants the highest affordable density. This is determined largely by the areal density trends. There is some variation in numbers of platters in a drive, but it is possible to build a drive of any chosen capacity for either market.

Data rate is proportional to spindle speed, areal density, and platter size. The data rate for the enterprise market tends to be higher than for personal storage, but higher spindle speeds cost more regardless of the interface used.

Fast seeks cost more and target the enterprise market. This includes larger magnets, better bearings, and stiffer actuators. The challenge is to rapidly find the target track (seek) and then to stay on track (servo) in spite of the harsh electrical and magnetic environment.

Protection from rotational vibration costs extra and targets markets where multiple drives sit next to each other. This includes better motors, top covers, stiffer actuators, and additional mass.

Better scheduling costs extra, requiring more code space, more memory for re-order queues and for algorithms. This is easier to do in the SCSI interface because it has traditionally had queueing and is more mature, but the implementation complexity would exist regardless of the interface used.

Fancier interface electronics cost extra. Because SCSI is richer and more complex, with more customer-modifiable options and host connectivity, it takes more electronics and more memory space. This is the only difference that truly arises solely from the choice of interface.

Finally, high reliability costs extra. It needs to be considered in every component and material choice along the way, as well as in the overall design. It also has to take into account the duty cycle targets for the expected workload and the expected environment.

6 Conclusions

The differences between enterprise and personal storage disc drives are significant. They derive from the different requirements of the respective markets and offer a range of choices to system designers. Simply separating the products by their external interface - ATA vs. SCSI - misses many of the internal details and design choices that will affect system performance. We have shown that the external interface chosen is one of the smallest contributors to overall performance. The performance and reliability characteristics of a drive are determined by the way the drive is designed - from the smallest mechanical and materials choices in the head-disc assembly, through the seek scheduling algorithms in the interface processing. In order to find the right features and design points for a particular application, the underlying trade-offs must be taken into account across a continuum of specific choices.

7 Acknowledgements

We thank Albertine Flora for help with the performance testing and benchmarking. The details of drive internals reported here are due to discussions with Zip Cotter, Neal Gunderson, and Jim Weispfenning. We thank Kevin Gomez and Mark Lutwyche for explaining some of the terminology. This analysis and comparison builds on previous work led by Heath Miller and Ed Skalko, with contribution from many others. We thank the anonymous reviewers and especially our shepherd, John Wilkes, for many detailed and helpful comments.

8 References

[Abramovitch96]	Abramovitch, D.Y. "Rejecting rotational disturbances on small disk drives using rotational accelerometers" *IFAC World Congress*, July 1996.
[Aruga01]	Aruga, K. "3.5-inch High-Performance Disk Drives for Enterprise Applications: AL-7 Series", *Fujitsu Scientific & Technical Journal* 37 (2), December 2001.
[Augustus01]	Augustus "Seagate Cheetah X15 36LP Review" *www.linuxhardware.org/features/01/09/10/065209.shtml*, LinuxHardware.org, September 2001.

[Blount01] Blount, W.C. "Fluid Dynamic Bearing Spindle Motors" *IBM Storage Systems Group - San Jose*, February 2001.

[Chung00] Chung, L., Gray, J., Worthington, B. and Horst, R. "Windows 2000 Disk IO Performance" *Technical Report MSR-TR-2000-55*, Microsoft Research, June 2000.

[CNET02] CNET Hardware Reviews. May 2002.

[Cole00] Cole, G. "Estimating Drive Reliability in Desktop Computers and Consumer Electronics Systems" *Technology Paper TP-338.1*, Seagate Technology, November 2000.

[Dominguez99] Dominguez, R. and Coligan, T. "SCSI vs. ATA: Interface Comparison" *Technology Brief*, Dell Computer, December 1999.

[Elerath00] Elerath, J.G. "Specifying Reliability in the Disk Drive Industry: No More MTBF's" *IEEE Annual Reliability and Maintainability Symposium*, January 2000.

[Hall00] Hall, J. "Seagate's AdvancedMultidrive System (SAMS) Rotational Vibration Feature" *Technology Paper TP-229D*, Seagate Technology, February 2000.

[Herbst97] Herbst, G. "IBM's Drive Temperature Indicator Processor (Drive-TIP) Helps Ensure High Drive Reliability" *White Paper*, International Business Machines Corp, October 1997.

[Hughes02] Hughes, G.F., Murray, J.F., Kreutz-Delgado, K. and Elkan, C. "Improved Disk Drive Failure Warnings" *IEEE Transactions on Reliability*, September 2002.

[IBM99a] IBM "IBM hard disk drive load/unload technology" *IBM Storage Systems Division*, May 1999.

[IBM99b] IBM "Higher reliability with IBM glass substrate disks" *IBM Storage Systems Division*, July 1999.

[IBM99c] IBM "Advanced Servo-Mechanical Design Facilitates Improved Performance and Reliability" *IBM Storage Systems Division*, November 1999.

[IBM01] IBM Research News "IBM's new magnetic hard-disk-drive media delays superparamagnetic effects" *www.research.ibm.com/resources/news/20010518_whitepaper.shtml*, February 2001.

[Kaczeus90] Kaczeus, S. "Disk reliability is a function of design as well as manufacture" *Computer Technology Review* 10 (9), Summer 1990.

[Mason00] Mason, H. "SCSI, the industry workhorse, is still working hard" *IEEE Computer*, December 2000.

[Matsumoto99] Matsumoto, C. "Diskcon abuzz over single-chip drives, home markets" *EE Times*, *eetimes.com/sys/news/OEG19990917S0013*, September 1999.

[Miura01] Miura, Y. "Information Storage for the Broadband Network Era - Fujitsu's Challenge in Hard Disk Drive Technology" *Fujitsu Scientific & Technical Journal* 37 (2), December 2001.

[Patel01] Patel, C.D., Bash, C.E., Belady, C., Stahl, L. and Sullivan, D. "Computational Fluid Dynamics Modeling of High Compute Density Data Centers to Assure System Inlet Air Specifications" *ASME International Electronic Packaging Technical Conference and Exhibition (IPACK '01)*, July 2001.

[Reinsel00] Reinsel, D. "180 Gigabytes in One Drive: Seagate's Barracuda 180" *IDC Bulletin*, November 2000.

[Richkus99] Richkus, R., Agogino, A.M., Yu, D., and Tang, D. "Virtual Disk Drive Design Game with Links to Math, Physics and Dissection Activities" *29th ASEE/IEEE Frontiers in Education Conference*, San Juan, Puerto Rico, November 1999. Online at *bits.me.berkeley.edu/mmcs/disk/disk.html*

[Seagate00] Seagate "Disk Drive Acoustics" *Technology Paper TP-296*, Seagate Technology, April 2000.

[Seagate02] Seagate Support "Disc Drive Encyclopedia" *www.seagate.com/support/disc*, Seagate Technology, August 2002.

[Talagala99] Talagala, N. and Patterson, D. "An Analysis of Error Behavior in a Large Storage System" *Technical Report UCB/CSD-99-1042*, University of California - Berkeley, February 1999.

[Vilsbeck02] Vilsbeck, C. "Gefahr: IDE-Festplatten im Dauereinsatz" *www.tecchannel.de/hardware/964/index.html*, tecCHANNEL.de, June 2002.

[Walker01] Walker, M. "Performance Media: Tweaking Magnetic Capabilities" *Technology Paper TP-577*, Seagate Technologies, August 2001.

[White01] White, B. Ng, W.T. and Hillyer, B.K. "Performance Comparison of IDE and SCSI Disks" *Technical Report*, Bell Labs - Lucent Technologies, January 2001.

[Worthington95] Worthington, B.L., Ganger, G.R., Patt, Y.N., Wilkes, J. "On-Line Extraction of SCSI Disk Drive Parameters" *SIGMETRICS*, May 1995.

[Yang99] Yang, J. and Sun, F. "A Comprehensive Review of Hard-Disk Drive Reliability" *IEEE Annual Reliability and Maintainability Symposium*, January 1999.

9 Appendix

The data in Table 7 shows multiple generations of drives from several manufacturers, including both ATA and SCSI interfaces. These numbers serve as a reference for the comparisons made in the paper. Detailed data is provided for all the drives discussed in the text, those mentioned in previous studies, and some recently released drives.

	iface	intro	cap	price	speed	seek	density	kbpi	ktpi	dia	int bw		ext bw	disks	cache
					rpm						calc	spec			
Quantum Atlas 10K	SCSI	1999	18 GB	-	10000	4.5 ms	3.4 Gb/in^2	256	13.0	3.3"	444	314 Mb/s	25 MB/s$^{@\&}$	6	2 MB
Maxtor Fireball lct 08	ATA	1999	26 GB	-	5400	9.5 ms	6.1 Gb/in^2	324	19.5	3.7"	343	257 Mb/s	19 MB/s$^{\&}$	3	512 KB
IBM UltraStar 36LZX	SCSI	1999	36 GB	-	10000	4.9 ms	7.0 Gb/in^2	352	20.0	3.0"	552	452 Mb/s	36 MB/s^	6	4 MB
Seagate Cheetah X15	SCSI	2000	18 GB	-	15000	3.9 ms	7.3 Gb/in^2	343	21.4	2.6"	689	508 Mb/s	40 MB/s$^{@}$	5	16 MB*
Quantum Atlas 10K II	SCSI	2000	18 GB	-	10000	4.7 ms	7.7 Gb/in^2	341	14.2	3.3"	591	478 Mb/s	-	3	8 MB
IBM UltraStar 36Z15	SCSI	2001	36 GB	$365	15000	4.1 ms	10.7 Gb/in^2	397	27.0	2.6"	798	647 Mb/s	53 MB/s$^{\%}$	6	4 MB
IBM DeskStar 75GXP	ATA	2000	30 GB	-	7200	8.5 ms	11.0 Gb/in^2	391	28.4	3.7"	551	444 Mb/s	37 MB/s$^{\wedge @}$	2	2 MB
IBM UltraStar 73LXZ	SCSI	2001	36 GB	$239	10000	4.9 ms	13.1 Gb/in^2	480	27.3	3.3"	832	690 Mb/s	57 MB/s$^{\%}$	3	4 MB
Seagate Barracuda 180	SCSI	2001	180 GB	$1369	7200	7.4 ms	15.0 Gb/in^2	490	31.2	3.7"	691	508 Mb/s	-	12	16 MB*
Fujitsu AL-7LX	SCSI	2001	36 GB	$369	15000	4.0 ms	15.8 Gb/in^2	450	35.0	2.7"	954	734 Mb/s	-	4	8 MB
Seagate Cheetah X15-36LP	SCSI	2001	36 GB	$395	15000	3.6 ms	17.5 Gb/in^2	482	38.0	2.6"	969	709 Mb/s	58 MB/s$^{@}$	4	8 MB
Seagate Cheetah 73LP	SCSI	2001	73 GB	-	10000	5.1 ms	18.4 Gb/in^2	485	38.0	3.3"	840	671 Mb/s	-	4	4 MB
Fujitsu AL-7LE	SCSI	2001	73 GB	$529	10000	5.0 ms	19.2 Gb/in^2	485	39.5	3.3"	838	673 Mb/s	-	4	8 MB
Maxtor DiamondMax D540X-4G	ATA	2001	160 GB	-	5400	12.0 ms	25.2 Gb/in^2	442	57.0	3.7"	467	347 Mb/s	38 MB/s$^{<}$	3	2 MB
IBM DeskStar 120GXP	ATA	2000	60 GB	$105	7200	8.5 ms	29.7 Gb/in^2	547	54.0	3.7"	771	592 Mb/s	48 MB/s$^{\%}$	2$^{\#}$	2 MB
IBM DeskStar 120GXP	ATA	2000	120 GB	-	7200	8.5 ms	29.7 Gb/in^2	547	54.0	3.7"	771	592 Mb/s	50 MB/s$^{<}$	3	2 MB
Seagate Barracuda IV	ATA	2001	80 GB	$125	7200	9.5 ms	31.3 Gb/in^2	540	58.0	3.7"	761	555 Mb/s	41 MB/s$^{\%}$	2	2 MB
Seagate Cheetah 10K.6	SCSI	2002	146 GB	$1139	10000	5.3 ms	34.0 Gb/in^2	570	64.0	3.3"	-	841 Mb/s	-	4	8 MB
Seagate Cheetah 15K.3	SCSI	2002	73 GB	$769	15000	4.0 ms	34.0 Gb/in^2	533	64.0	2.5"	1071	891 Mb/s	-	4	8 MB
Western Digital Caviar WD1200	ATA	2002	120 GB	$179	7200	10.9 ms	-	-	-	3.7"	-	736 Mb/s	50 MB/s$^{<}$	2	8 MB*
Seagate Barracuda V	ATA	2002	120 GB	$185	7200	10.5 ms	42.2 Gb/in^2	542	78.0	3.7"	764	570 Mb/s	44 MB/s$^{\%}$	2	8 MB*
Western Digital Caviar WD2000	ATA	2002	200 GB	$359	7200	10.9 ms	45.0 Gb/in^2	-	-	3.7"	-	525 Mb/s	-	3	8 MB*

Table 7: Comparison of multiple drive generations and manufacturers. All numbers are from manufacturer specifications or product manuals, except where noted. Prices for drives still being sold in August 2002 are from dirtcheapdrives.com. Seek times are for average seek. All values for density and bandwidth are maximums (outer diameter). Internal bandwidth is calculated from the rpm, Kbpi, and disc diameter values and provided for comparison to the published values. $^{\#}$the 60 GB version of the DeskStar 120 has 2 disks, but only 3 heads, one side remains unused $^{@}$as measured by Linuxhardware.org [Augustus01] ^as measured at Bell Labs [White01] $^{\&}$as measured under Windows 2000 [Chung00] $^{\%}$according to the published specifications, not measured numbers $^{<}$as measured by CNET Hardware [CNET02] *option, the default cache size is 2 MB

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Atos Origin B.V. ❖ Freshwater Software ❖
❖ Interhack Corporation ❖
❖ The Measurement Factory ❖ Microsoft Research ❖
❖ Sendmail, Inc. ❖ Sun Microsystems, Inc. ❖
❖ Sybase, Inc. ❖ UUNET Technologies, Inc. ❖
❖ Veritas Software ❖ Ximian, Inc. ❖

SAGE Supporting Members

❖ Aptitune Corporation ❖ Certainty Solutions ❖
❖ Collective Technologies ❖ Freshwater Software ❖
❖ Microsoft Research ❖ Ripe NCC ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-07-2